2-2006

FPGA implementation of Reed Solomon codec for 40Gbps Forward Error Correction in optical networks

Kenny Chung Chung Wai

FPGA implementation of Reed Solomon Codec for 40Gbps Forward Error Correction in Optical Networks

by

Kenny Chung Chung Wai

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Shanchieh Jay Yang
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

Feb 2006

Approved By:

Shanchieh Jay Yang

Dr. Shanchieh Jay Yang
Assistant Professor, Department of Computer Engineering
Primary Adviser

Marcin Lukowiak

Dr. Marcin Lukowiak
Visiting Assistant Professor, Department of Computer Engineering

Doug Bush

Doug Bush
Director of IP Development, Xelic Inc.

Mark Garbosky

Mark Grabosky
Director of Design Methodology, Xelic Inc.

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: FPGA implementation of Reed Solomon Codec for 40Gbps Forward Error Correction in Optical Networks

I, Kenny Chung Chung Wai, hereby grant permission to the Wallace Memorial Library reproduce my thesis in whole or part.

Kenny Chung Chung Wai

Kenny Chung Chung Wai

3-8-06.

Date

Dedication

I would like to dedicate this thesis to my parents, my brother's and my sister's family.

Acknowledgments

I would like to acknowledge Dr. Yang for his guidance and mentoring, Mr. Grabosky, Mr. Bush and Dr. Lukowiak for serving as advisors on the thesis committee. I would also like to thank Xelic Inc., relatives and friends for their support.

Abstract

Reed-Solomon error correcting codes (RS codes) are widely used in communication and data storage systems to recover data from possible errors that occur during data transfer. A growing application of RS codes is Forward Error Correction (FEC) in the Optical Network (OTN G.709), which uses RS(255,239) to support the OTU-3 (43.018 Gbps) standard. There have been considerable efforts in the area of RS architecture for ASIC implementation. However, there appears to be little reported work on efficient RS codec (encoder and decoder) for Field Programmable Gate Arrays (FPGAs), which has increasing interests in industry.

This thesis investigates the implementation and design methodology of the RS(255,239) codec on FPGAs. A portable VHDL code is developed and synthesized for Xilinx's Virtex4 and Altera's StratixII. The FPGA architectures are analyzed and the required design methodologies are adopted to efficiently utilize the available resources. Unfortunately, due to the fixed size of FPGA devices, the RS decoder is not only constrained by the required timing of the system, but also by the size of the targeted device. This research will facilitate the decision-making process for selecting a reconfigurable device for a RS decoder, implementing the Berlekamp-Massey Algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use and demand for optical communications have grown tremendously for applications transmitting voice, data or video over short and long haul distances. The high bandwidth requirements spawn the introduction of an Optical Transport Network (OTN) protocol, defined in the ITU-T G.709 specification [33]. This protocol describes three interfaces: OTU-1, OTU-2 and OTU-3 at rates of 2.666 Gbps, 10.709 Gbps and 43.018 Gbps, respectively. Reed-Solomon (RS) Forward Error Correction (FEC) is included as part of the standard to increase reliability by correcting errors that may be introduced. There have been considerable efforts [28] [31] [15] in optimizing RS architectures for ASIC implementation based on the Euclidean [32] and Berlekamp-Massey algorithm [5]. However, it is unclear if the performance of RS codecs on ASICs can be obtained on FPGAs since there is relatively little work on how to optimize RS codecs for FPGA.

The two leading FGPA manufacturers, Xilinx and Altera, provide their own versions of RS codec targeting their devices. Xilinx uses the Euclidean algorithm whereas Altera uses the Berlekamp-Massey algorithm to implement the Key Equation [4]. It is unclear why they implemented different decoding algorithms and how the FPGA architectures affect the implementation since the implementation is not in the public domain.

The goal of this thesis was to develop a generic VHDL model of the Reed-Solomon codec that can be synthesized to Xilinx's Virtex4[30] and Altera's StratixII[23]. The RS codec was optimized to operate at the required speed of OTU-3. The performance of the RS codec on Virtex4 and StratixII was measured and compared to explore the impact of

FPGA architecture on VHDL model of RS codec. This contribution will facilitate the decision-making of choosing a reconfigurable device for a RS codec implementation.

1.1 Background

1.1.1 Forward Error Correction (FEC) methods

More than 80% of the world's long distance and data transfer is carried over fiber optics, ranging from global networks to desktop computers. To maintain the reliability of the data traveling at high speed, various Forward Error Correction (FEC) techniques have been proposed to correct errors introduced through transmitting over a noisy channel. Prior to transmission, the FEC encoder introduces redundancy to the data. On receiving, the receiver will detect and correct up to a maximum number of errors, and such limit depends on the algorithm used. The general concept of FEC is illustrated in Figure 1.1. FEC

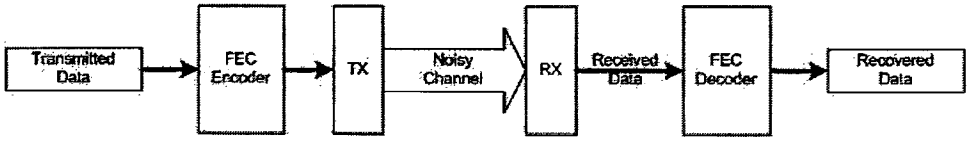

Figure 1.1: Forward Error Correction Concept

algorithms can be categorized into two main groups:

- **Convolution Coding**

  A convolution code is a type of error-correction codes in which an entire stream of data is divided into fixed-size symbols, each containing m bits. Using a predetermined algorithm, k consecutive symbols are encoded to form a codeword of n bits where $n \geq m$. Therefore, each codeword depends on k previous symbols. The codewords are also concatenated to form a continuous and theoretically infinite stream of code symbols. This is one of the main reasons for using convolutional code in radio and satellite communications [8]. A commonly used convolutional code is based on the Viterbi algorithm [34].

- **Block Coding**

  A block code is another type of error-correcting codes in which a stream of data is divided into fixed-size blocks, each containing k information symbols of predetermined size. Each block is encoded to form a codeword of size n symbols. A codeword, therefore, contains k information symbols, appended with $n - k$ parity symbols. One well-known block code is the ubiquitous Reed-Solomon codes, widely used in many applications for data storage and data transfer [36].

In 1993, Berrou, Glavieux and Thitimajshima introduced the Turbo codes [2]. This high-performance error-correcting code operates to within 0.7dB of Shannon's limit [12], which is the theoretical maximum information transfer rate of a channel. It uses two or more convolution codes and an interleaver to produce a block code. Despite the high bandwidth efficiency of Turbo codes, RS codes are still widely used in applications such as Asynchronous Transfer Mode (ATM) networks and Optical network (OTN)[33]

1.1.2 OTN G.709

A growing application of the RS code, specifically RS(255,239), is for FEC in OTN G.709 [20] because of its relatively high error correction capability and low error burst sensitivity. The OTN G.709 is an optical network protocol that provides higher backbone bandwidth. The OTN frame structure can be referred to as Optical channel Transport Unit (OTUk)

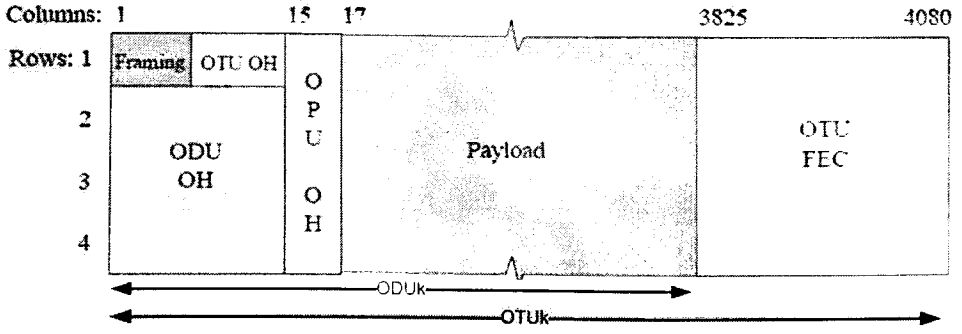

Figure 1.2: OTUk Frame Structure

and is based on the Optical channel Data Unit (ODUk) frame structure appended with the OTUk Forward Error Correction (FEC) as shown in Figure 1.2. The ODUk contains the ODUk overhead area and the Optical channel Payload Unit (OPUk), and the FEC contains parity symbols. The OTUk frame structure contains four rows and each row is made by interleaving 16 RS(255,239) codewords as shown in Figure 1.3. If no FEC is used, all the

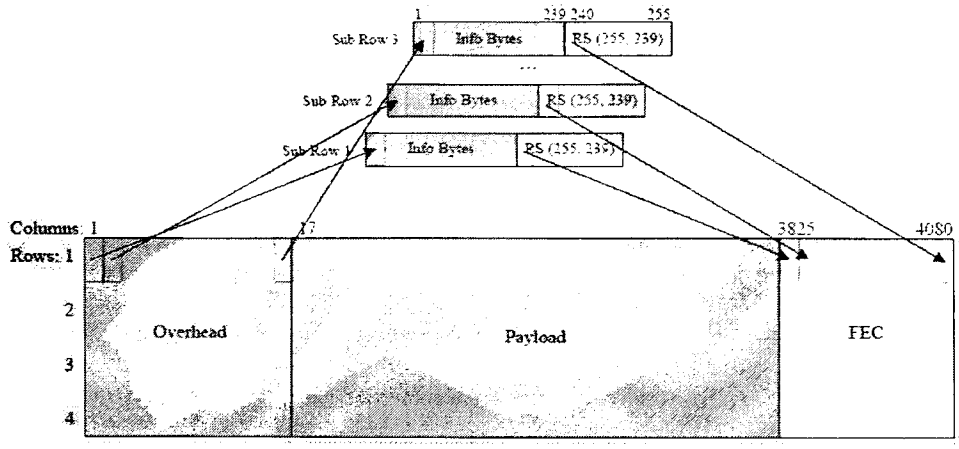

Figure 1.3: OTUk based on ODUk and FEC [27]

4 x 256 bytes in OTUk FEC section are set to zero. Three standard rates are defined in the OTN G.709 specification [20]:

- OTU-1: 2.666 Gbit/s
- OTU-2: 10.709 Gbit/s
- OTU-3: 43.018 Gbit/s

Currently, the OTU-1 and OTU-2 are primarily used to transparently carry Synchronous Optical Network (SONET) frames, Ethernet frames, etc. However, the structures supporting the OTU-3 rate have not been fully released. The OTU-3 standard has a serial transfer rate of 43.018 Gbit/s. To achieve this rate, a data bus of 32 x 8 bit wide is typically used with clock speed of 168.05 MHz.

1.1.3 Reed-Solomon Codes

The class of RS codes is a subclass of the Bose-Chaudhuri-Hocquenghem (BCH) codes, discovered in the late 50s [4]. The class of BCH codes belongs to the block codes family and is able to correct multiple error correcting codes. The BCH codes can be divided into two subclasses: binary BCH and Reed-Solomon codes. In 1960, two members of the MIT Lincoln Lab, Irving S. Reed and Gustave Solomon, published a seminar paper [29] and marked the beginning of the RS codes. The seminar paper essentially showed that a vector space of m dimension can be mapped over a finite field K into a vector space of higher dimension n over the same field, and assuming no more than $(n-m)/2$ errors occur during transmission in the vector space of dimension n, there exists a decoding procedure which recovers the errors completely. In 1963, Peterson, Gorenstein and Zierler (PGZ) presented the first algorithm that explicitly described a decoding algorithm [35] of the RS codes. PGZ's decoding algorithm, however, as the was limited by the size of the codeword because of the matrix inversions needed to calculate the locations and the magnitudes of the errors. By the end of the 60s, Berlekamp [1] and Massey [26] combined their effort to provide the Berlekamp-Massey (BM) algorithm, the first efficient algorithm to decode the RS codes. Instead of computing matrix inversion, the Berlekamp-Massey (BM) algorithm solved a "Key Equation" to locate and evaluate the errors. In 1975, Sugiyama [32] proved that the traditional Euclidean algorithm could also be used to solve the "Key Equation," thus decoding BCH and Reed-Solomon codes. In 1995, Fitzpatrick [9] introduced a new decoding algorithm and claimed that his algorithm utilized fewer finite field multipliers than the Berlekamp-Massey's algorithm. A year later, in 1996, Blackburn and Chambers [3] pointed out a flaw in the claim made by Fitzpatrick. After analyzing both algorithms, they found out that the version of Berlekamp-Massey algorithm that Fitzpatrick had used to benchmark against his algorithm was not the most efficient implementation. After comparing the complexity of both algorithm, they concluded that Fitzpatrick's algorithm was as efficient as the BM's algorithm. The BM and the Euclidean algorithm are the two most commonly-used decoding algorithms for RS codes.

The Berlekamp-Massey algorithm is an efficient method in decoding the Reed-Solomon codes [5]. Consequently, much research has been done to optimize for implementation. Sarwate [31] proposed a high-speed architecture in which a single array of processors are used to compute both the error-locator and the error-evaluator polynomials. Sarwate's architecture required approximately 25% fewer multipliers and a simpler control structure than the architecture based on the Euclidean algorithm. Raghupathy [28] modified the BM decoding algorithm to not only increase the speed, but also obtain a low-power architecture. The results of his implementation indicated a power reduction of about 40% and a speed-up of 1.34 compared to a normal design without his proposed modifications. Another way to improve the speed of the BM algorithm was to use the "division-free" algorithm, proposed by Dinh [7]. To verify the functionality of the decoder, Dinh compiled his HDL code to an Altera FPGA device, EPF10K200A. The designed decoder occupied 12,745 logic cells and operated at 12MHz. When synthesized using a 0.18μ CMOS technology, the same design operated at the clock speed ranging between 125MHz to 250MHz.

Similar to the BM algorithm, optimized architecture and implementation of the Euclidean algorithm have also been the subjects for a number of research papers. A lot of effort has been invested to improve the architecture of the Euclidean algorithm. Lee has been a major contributor to the evolution of RS decoding based on the Euclidean algorithm [16]. In 2001, he proposed a high speed design of the Reed-Solomon decoder [15] using a modified version of the Euclidean algorithm. In 2003, he implemented an area-efficient Euclidean Block for Reed-Solomon decoder [17] which targeted on the 0.13μ CMOS technology. In the same year, Lee and Azam presented a novel pipelined recursive modified Euclidean algorithm block for a low-complexity, high-speed RS decoder [19].

All of the afore-mentioned works on both the BM and the Euclidean algorithm were targeted to CMOS technologies. Among the few papers discussing implementations of the RS codec on reconfigurable devices, Flocke [10] introduced a highly parameterizable RS-decoder for FPGAs in 2005. His implementation, based on the inversionless Berlekamp Algorithm, was tested on Altera APEX 20KE device and achieved a high throughput rate

of 1.3 Gbit/s. A case study [13] about using the run-time reconfigurability (pRTR) feature available on VIRTEX FPGAs was published in 2002. The author referred to the design of a RS decoder as an example to go over the methodology and design flow for VIRTEX FPGAs which enabled the user to implement large designs into a moderately sized FPGA. Unfortunately, none of these research reported which decoding algorithm is appropriate in decoding RS codes for FPGA devices.

1.1.4 ASIC vs FPGA

FPGA, Field Programmable Gate Array, denotes an integrated circuit that is configured in the field, whereas ASIC, Application Specific Integrated Circuit, denotes an integrated circuit that is fully customized to the requirements of a given application. Before making a choice between ASIC and FPGA for a specific design, several parameters must be considered.

FPGA has become very popular in industry because of its short development time. FPGAs are available off-the-shelf whereas for ASICs, a typical lead time between eight to sixteen weeks is needed to get the design out of the factory. The designers who use FPGAs have the real silicon to test the implementation instead of testing through simulation only. The FPGAs can be reconfigured as many times as needed during development. There is no NRE cost for FPGA, but the unit cost of one FPGA is higher than that of ASIC. Depending on the production volume, ASIC design may still be chosen over FPGA because of the high yield of an optimized architecture.

Designing in ASIC and FPGA requires different design methodologies. Designing for ASICs, the designer needs to focus on how to optimize the architecture of the system so that it consumes the least area and runs at the required speed. Whereas designing for FPGAs, not only the designer needs to focus on optimizing the system, but he also needs to understand the architecture of the chosen FPGA device so that he can make use of the available resources and efficiently utilize these resources. Since ASIC and FPGA are completely two different technologies, a design created for an ASIC may not work well on an FPGA. Most

works in the field of RS decoder are targeted to ASIC and there is no related work on how the algorithms used RS decoder are dependent on FPGA architectures. Altera and Xilinx are two main FPGA manufacturers and it is a known fact that FPGA architecture is specific to vendors. Because of the difference in architecture, it is hard to predict the physical behavior of the RS codec from one FPGA family to another. Both companies mentioned above have their Intellectual Property (IP) cores for RS codec. However, Xilinx implements the Euclidean algorithm, and Altera implements the Berlekamp-Massey algorithm for their RS decoders.

1.2 Thesis Contribution

The goal of this thesis work was to investigate the effect of FPGA architecture on RS codec implementation. A portable Reed-Solomon codec was designed in VHDL and its performance was measured on Xilinx's Virtex4 [30] and Altera's StratixII [23]. The implemented RS codes was specified by RS(255, 239), used for error correction in OTN G.709 [33]. The resource utilization and speed of the two FPGA devices were the targeted performance metrics for the analysis.

One dominant factor which limits the performance of the decoder is the architectural features of the FPGAs. The VHDL code was compiled toward both the Virtex4 and StratixII FPGA families, and the synthesis results lead to a better understanding of the impact of FPGA architecture on RS codec performance. The performance of the implemented codec achieved the required speed to implement FEC in OTU-3. However, such result did not repeat itself on Xilinx's Virtex4. It was found that Altera's architecture favors Berlekamp-Massey's algorithm to solve the key equation. Research and development is underway to confirm the performance of Euclidean's algorithm on Xilinx's Virtex4.

This thesis is organized as follows. Chapter 2 describes the Reed-Solomon codes and different algorithms of the Reed-Solomon decoders. Chapter 3 describes the design and implementation of the generic Reed-Solomon decoder. The results of the implementation

are presented and analyzed in Chapter 4. Chapter 5 will conclude the thesis and highlight the plans for future work.

Chapter 2

Reed-Solomon

2.1 What is a Reed-Solomon(RS) code?

A Reed-Solomon code [29] is the mapping from a vector space of m dimension over a finite field K into a vector space of higher dimension n over the same field, and assuming no more than $(n-m)/2$ errors occur during transmission in the vector space of dimension n, there exists a decoding procedure which recovers the errors completely. A Reed-Solomon code is a block code and is specified as RS(n,k) as shown in Figure 2.1. n defines the size of the

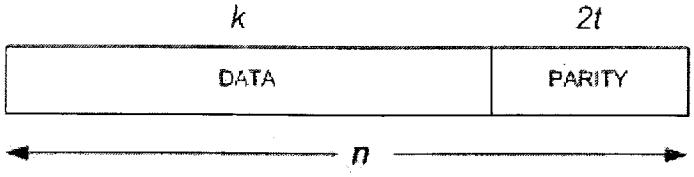

Figure 2.1: the structure of a RS codeword

codeword in the unit of symbols, k is the number of data symbols and $2t = n - k$ is the number of parity symbols. Each symbol contains s bits, where

$$s = log_2(n+1) \text{ or } n = 2^s - 1 \tag{2.1}$$

The relationship implies that the use of s-bit symbols allows for a maximum of $2^s - 1$ distinct symbols in one codeword, excluding the one with all zeros. The maximum number of symbol errors that the RS code can correct is given by t, half the size of the parity. For

example, consider RS(255,239) defined for the optical network specification, OTN G.709 [33]. The size n is 255 symbols, the number of data symbols k is 239 symbols. The maximum number of symbol errors that RS(255,239) decoder can correct is, therefore, 8 symbols, where each symbol contains 8 bits.

2.2 How does RS code work?

The Reed-Solomon code is defined in the Galois field [4]. A Galois field is a finite field that contains a set of nonzero elements forming a cyclic group under multiplication. Each nonzero element can be expressed as a power of a primitive element, α, of the field.

The Galois field GF(2^s) is constructed based on a primitive polynomial, $p(z)$, which determines the pattern of the sequence. For example, consider the RS(7,3) with the primitive polynomial $p(z) = z^3 + z + 1$. n is 7 and each symbol contains $\log_2(n+1) = 3$ bits. A Galois field, GF(2^3), with the specified primitive polynomial is constructed as shown in Table 2.1.

Exponent	Polynomial	Binary
α^0	1	001
α^1	z	010
α^2	z^2	100
α^3	$z + 1$	011
α^4	$z^2 + z$	110
α^5	$z^2 + z + 1$	111
α^6	$z^2 + 1$	101
$\alpha^7 = \alpha^0$	1	001
$\alpha^8 = \alpha^1$	z	010

Table 2.1: Galois field, GF(8), with $p(z) = z^3 + z + 1$

An addition of two elements in the Galois field is simply the exclusive-OR (XOR) operation. For example

$$\begin{aligned} \alpha^5 + \alpha^6 &= (z^2 + z + 1) \oplus (z^2 + 1) \\ &= z = \alpha^1 \end{aligned}$$

However, a multiplication in the Galois field is more complex than the standard arithmetic. It is the multiplication modulo the primitive polynomial $p(z)$. For example,

$$\begin{aligned} \alpha^5 \bullet \alpha^6 &= \alpha^5 \bullet \alpha^6 \; mod \; p(z) \\ &= (z^2 + z + 1) \bullet (z^2 + 1) \; mod \; (z^3 + z + 1) \\ &= z^2 + z = \alpha^4 \end{aligned}$$

A stream of data | 010 | 100 | 000 | that needs to be transmitted over the channel can be represented as a transmitted message polynomial $m(x)$(2.2).

$$m(x) = \alpha x^2 + \alpha^2 x. \tag{2.2}$$

Table 2.2 defines the notation for the polynomials that will be used to explain the formation and decoding of the RS code in the following sections

Item	Term Definition
$c(x)$	transmitted codeword
$m(x)$	transmitted message
$g(x)$	generator polynomial
$e(x)$	error polynomial
$r(x)$	received codeword
$S(x)$	syndrome polynomial
$\Omega(x)$	error evaluator polynomial
$\Lambda(x)$	error locator polynomial
m_0	a natural number
e	number of errors

Table 2.2: Term Definition

2.2.1 Encoder

Encoding the RS code is basically mapping a message of dimension k into a codeword of dimension n. The mapping is done using a generator polynomial (2.3) of dimension $2t$. One encoding scheme is known as non-systematic encoding, which is a Galois Field

multiplication between the transmitted message and the generated polynomial as shown in (2.3).

$$c(x) = m(x)g(x) \tag{2.3}$$

where $g(x)$ is the generator polynomial of degree $2t$ and is given by

$$g(x) = \prod_{i=m_0}^{m_0+2t-1} (x + \alpha^i) \tag{2.4}$$

m_0 is a natural number and defines the coefficients of the generator polynomial. The encoded codeword contains the original message $m(x)$ encrypted with $g(x)$. This encoding scheme is not very practical because at the receiving side, $m(x)$ is recovered when dividing $c(x)$ by $g(x)$ using polynomial long division, which requires additional computations.

An another encoding scheme was proposed and is known as systematic encoding as shown in (2.5).

$$c(x) = m(x)X^{2t} + m(x)X^{2t} \bmod g(x) \tag{2.5}$$

The transmitted codeword is encoded in a way that the transmitted message $m(x)$ appear as the first k symbols, appended with $2t$ parity symbols. At the receiving side, the message $m(x)$ is recovered by reading only the first k symbols.

Encoding Example

The message (2.2) is encoded with the generator polynomial defined as

$$\begin{aligned} g(x) &= \prod_{i=0}^{3} (x + \alpha^i) \\ &= x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^5 x + \alpha^6 \end{aligned}$$

$m(x)$ is first shifted by four symbols to the left and the result is divided by $g(x)$ as shown below

$$\begin{aligned} m(x)X^4 &= \alpha x^6 + \alpha^2 x^5 \\ m(x)X^4 \bmod g(x) &= \alpha x^6 + \alpha^2 x^5 \bmod x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^5 x + \alpha^6 \\ &= \alpha^3 x^2 + \alpha^5 x + \alpha \end{aligned}$$

Using (2.5), the resulting codeword is then given as

$$c(x) = \alpha x^6 + \alpha^2 x^5 + \alpha^3 x^2 + \alpha^5 x + \alpha \tag{2.6}$$

and can also be represented as | 010 | 100 | 000 | 000 | 011 | 111 | 010 | in binary. After decoding the received codeword, the transmitted stream of data is easily recovered by extracting the first *three* symbols from the codeword.

2.2.2 Decoder

Figure 2.2 shows the multiple approaches to decode the received data as described by Raghupathy and Liu[28]. A decoding algorithm is typically chosen according to a specific application. A commonly used RS decoding scheme for the OTN G.709 is shown in Figure 2.3, and because of its wide acceptance, this thesis work will focus on implementing this specific decoding scheme.

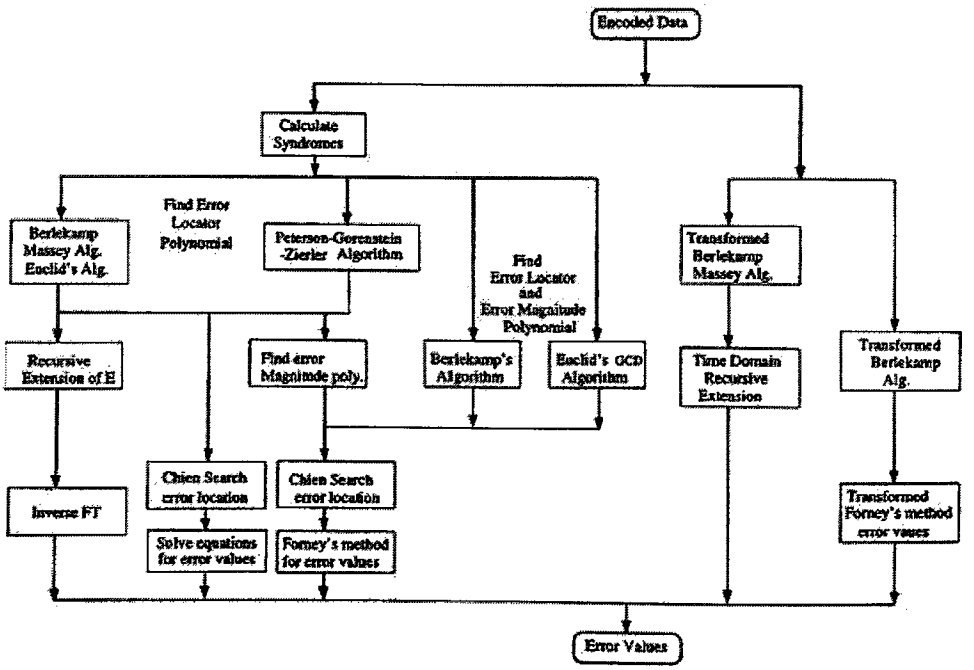

Figure 2.2: Different Decoding Techniques for Reed-Solomon codes [28]

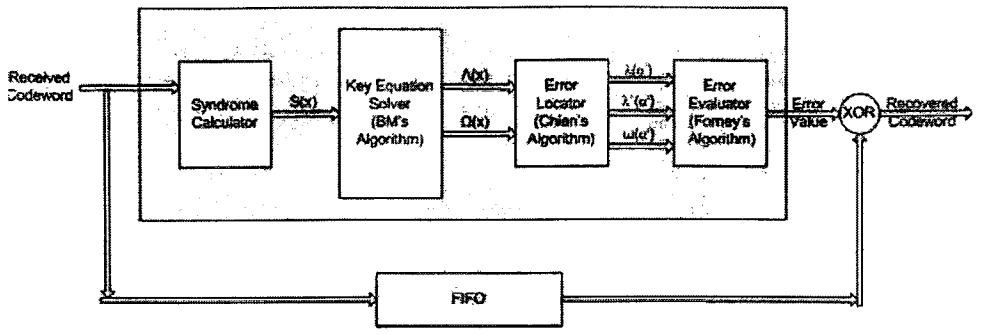

Figure 2.3: Decoding scheme to be implemented in OTN G.709

The received codeword is represented as a function of the transmitted codeword polynomial $c(x)$ and the error polynomial $e(x)$ as shown in (2.7).

$$r(x) = c(x) + e(x) \tag{2.7}$$

where $e(x)$ is the error introduced during transmission. The first functional block of the decoder in Figure 2.3 detects if the received codeword is erroneous. The Syndrome Calculator generates a syndrome polynomial $S(x)$ for every received codeword. $S(x)$ is used by the "Key Equation Solver" [4] to generate an error locator polynomial $\Lambda(x)$ and an error evaluator polynomial $\Omega(x)$. $\Lambda(x)$ is then used by the Error Locator to search the positions of the errors. Once the locations of the errors are found, the Error Evaluator generates the error polynomial, $e(x)$. The transmitted codeword, $c(x)$, is then recovered when the received message $r(x)$ is added to the error polynomial $e(x)$ as shown in (2.8).

$$c(x) = r(x) + e(x) = c(x) + e(x) + e(x) = c(x) \tag{2.8}$$

Syndrome Calculator

The transmitted codeword $c(x)$ is a multiple of the generated polynomial $g(x)$ and therefore, $c(x)$ is divisible by each single root of $g(x)$. Any error introduced in the transmitted codeword results in a remainder when individual root of $g(x)$ divides the receiving codeword $r(x)$. The sum of the remainders is referred as the syndrome polynomial $S(x)$, and is

defined in (2.9).

$$S(x) = \sum_{i=m_0}^{m_0+2t-1} r(\alpha^i),\ m_0 \leq i \leq m_0 + 2t - 1 \tag{2.9}$$

If $S(x) = 0$, the received codeword is error free. Otherwise, the syndrome polynomial is processed by the Key Equation Solver functional block.

Key Equation Solver

The objective of the Key Equation Solver is to solve an equation that describes the relationship between the syndrome polynomial $S(x)$, the error locator polynomial, $\Lambda(x)$, and the error evaluator polynomial $\Omega(x)$.

$$\Lambda(x)S(x) = \Omega(x) \bmod x^{2t} \tag{2.10}$$

where $\Lambda(x)$ and $\Omega(x)$ may be represented in the general form shown in (2.11) and (2.12), respectively.

$$\Lambda(x) = \prod_{j=1}^{e}(1 - X_j x) = 1 + \lambda_1 x + \lambda_2 x^2 + ... + \lambda_e x^e \tag{2.11}$$

$$\Omega(x) = \sum_{i=1}^{e} Y_i X_i \prod_{j=1, j\neq i}^{e}(1 - X_j x) = \omega_0 + \omega_1 x + \omega_2 x^2 + ... + \omega_{e-1} x^{e-1} \tag{2.12}$$

The error locator polynomial, $\Lambda(x)$, has a degree of $e \leq t$ and has as its roots the inverses of the e error locators $\{X_j\}$. The error evaluator polynomial, $\Omega(x)$ has degree at most $e - 1$ to determine the magnitude of e errors.

This is the most complex block to be designed and implemented. There are different algorithms that have been used to implement this component. However, most of these algorithms are optimized only for ASIC design. In this thesis, two commonly used algorithms are considered to implement the Key Equation block: the Berlekamp-Massey [4] and the Euclidean [18]. These algorithms are discussed in Section 2.4.

Chien Search Error Location

The locations of the errors are determined based on the error locator polynomial (2.11). Each α^j for $m_0 \leq j \leq m_0 + 2t - 1$ is plugged into (2.11). If $\Lambda(\alpha^j) = 0$, the inverse of α^j is calculated to be $\alpha^{-j} = \alpha^c$, and the location of an error be indicated by the exponent c. This process is known as the Chien search algorithm.

Forney's method for error values

In this block, the error evaluator polynomial $\Omega(x)$ is computed to find the corresponding error value at each error location. The output of this block is the error polynomial $e(x)$ which is a polynomial representing the value and location of the errors.

$$e_{i_k} = \frac{X_k \Omega(X_k^{-1})}{\Lambda'(X_k^{-1})} \tag{2.13}$$

The syndrome Calculator, Chien's search [6] and Forney's method [11] are relatively simple to implement compared to the Key Equation Solver. There exist different algorithms to implement the Key Equation Solver as discussed in the following section.

2.3 Algorithm Analysis

In 1960, Peterson's algorithm was the first explicit description of a decoding algorithm for binary BCH codes. One year after, Gorenstein and Zierler extrapolated Peterson's decoding algorithm to illustrate the decoding scheme of the Reed-Solomon codes. However, the GZP decoding algorithm was limited by the size of the code because of the need for matrix inversions to calculate the error-locations and magnitudes. Five years later, in 1965, marked the introduction of the Berlekamp-Massey algorithm. The Berlekamp and Massey's approach was to solve the "Key Equation" to locate and evaluate the errors. In 1975, it was proven that the Euclidean algorithm can also be used to solve the "Key Equation" in decoding the Reed-Solomon codes.

2.3.1 GZP's algorithm

The first step in decoding Reed-Solomon codes is to calculate the syndrome polynomial. A syndrome polynomial, $S(x)$, represents the error pattern of the received codeword, $r(x)$, and is used to generate the error-locator and error-evaluator polynomials.

The error pattern, S_j where $0 \leq j \leq 2t-1$, is formed by evaluating the received polynomial $r(x)$ at α^j, and is defined as

$$S_j = c(\alpha^j) + e(\alpha^j) = e(\alpha^j) \tag{2.14}$$

To streamline (2.14), the error-magnitude, e_{i_l}, is replaced by Y_l and the error-location, α^{i_l} is replaced by X_l. i_l is the actual location of the lth error, and X_l is the field element associated with this location. The syndromes are then given as

$$\begin{aligned} S_0 &= Y_1X_1^0 + Y_2X_2^0 + ... + Y_\nu X_\nu^0 \\ S_1 &= Y_1X_1^1 + Y_2X_2^1 + ... + Y_\nu X_\nu^1 \\ S_2 &= Y_1X_1^2 + Y_2X_2^2 + ... + Y_\nu X_\nu^2 \\ &\vdots \\ S_{2t-1} &= Y_1X_1^{2t-1} + Y_2X_2^{2t-1} + ... + Y_\nu X_\nu^{2t-1} \end{aligned} \tag{2.15}$$

This set of systematic nonlinear equations has at least one solution, but it is very hard to be directly solved. Therefore, (2.15) needs to be reduced to a set of linear functions. Consider the error-locator polynomial,

$$\Lambda(x) = \lambda_\nu x^\nu + \lambda_{\nu-1}x^{\nu-1} + ... + \lambda_1 x + 1 \tag{2.16}$$

defined to be the polynomial whose zeros are the inverses of the error location X_l^{-1} for $l = 1, ..., \nu$ as shown in (2.17)

$$\Lambda(x) = (1 - xX_1)(1 - xX_2)...(1 - xX_\nu) \tag{2.17}$$

The coefficients of $\Lambda(x)$ in (2.16) allows us to find the zeros of $\Lambda(x)$ in (2.17), thus leading to the error locations. We, therefore, need to calculate $\lambda_1, ..., \lambda_\nu$, by first setting

(2.16) = (2.17). Then, both sides are multiplied by $Y_l X_l^{j+\nu}$, and x is replaced by X_l^{-1} to give

$$\begin{aligned} Y_l X_l^{j+\nu}[1 + \lambda_1 X_l^{-1} + \lambda_2 X_l^{-2} + ... + \lambda_{\nu-1} X_l^{-(\nu-1)} + \lambda_\nu X_l^{-\nu}] &= 0 \\ Y_l[X_l^{j+\nu} + \lambda_1 X_l^{j+\nu-1} + ... + \lambda_\nu X_l^j] &= 0 \end{aligned} \quad (2.18)$$

Since (2.18) is valid for each l, these equations are summed up from $l = 1$ to $l = \nu$, giving for each j,

$$\begin{aligned} \sum_{l=1}^{\nu} Y_l[X_l^{j+\nu} + \lambda_1 X_l^{j+\nu-1} + ... + \lambda_\nu X_l^j] &= 0 \\ \sum_{l=1}^{\nu} Y_l X_l^{j+\nu} + \lambda_1 \sum_{l=1}^{\nu} Y_l X_l^{j+\nu-1} + ... + \lambda_\nu \sum_{l=1}^{\nu} Y_l X_l^j &= 0 \end{aligned} \quad (2.19)$$

The individual sums are equivalent to the syndromes, and (2.19) can be expressed as

$$\begin{aligned} S_{j+\nu} + \lambda_1 S_{j+\nu-1} + ... + \lambda_\nu S_j &= 0 \\ \lambda_1 S_{j+\nu-1} + \lambda_2 S_{j+\nu-2} + ... + \lambda_\nu S_j &= -S_{j+\nu} \end{aligned} \quad (2.20)$$

(2.20) is the set of linear equations relating the syndromes to the coefficients of $\Lambda(x)$ and can be expressed in a matrix form as shown in (2.21).

$$\begin{pmatrix} S_0 & S_1 & S_2 & ... & S_{\nu-2} & S_{\nu-1} \\ S_1 & S_2 & S_3 & ... & S_{\nu-1} & S_\nu \\ S_2 & S_3 & S_4 & ... & S_\nu & S_{\nu+1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ S_{\nu-1} & S_\nu & S_{\nu+1} & ... & S_{2\nu-3} & S_{2\nu-2} \end{pmatrix} \begin{pmatrix} \lambda_\nu \\ \lambda_{\nu-1} \\ \lambda_{\nu-2} \\ \vdots \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} -S_\nu \\ -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu-1} \end{pmatrix} \quad (2.21)$$

$$\text{Let M be} \begin{pmatrix} S_0 & S_1 & S_2 & ... & S_{\nu-2} & S_{\nu-1} \\ S_1 & S_2 & S_3 & ... & S_{\nu-1} & S_\nu \\ S_2 & S_3 & S_4 & ... & S_\nu & S_{\nu+1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ S_{\nu-1} & S_\nu & S_{\nu+1} & ... & S_{2\nu-3} & S_{2\nu-2} \end{pmatrix} \quad (2.22)$$

The number of errors, e, is set to be the maximum number of correctable errors, $\nu = t$. If det(M) = 0, the number of errors is less than ν and the matrix, M, is reduced by deleting the rightmost column and the bottom row. ν is decremented by one and the determinant of the reduced matrix is calculated. This exercise is repeated until det(M) $\neq 0$. When the determinant is nonzero, the number of errors, $e = \nu$ and the error-locators are evaluated in (2.23).

$$\begin{pmatrix} \lambda_\nu \\ \lambda_{\nu-1} \\ \lambda_{\nu-2} \\ \vdots \\ \lambda_1 \end{pmatrix} = M^{-1} \begin{pmatrix} -S_\nu \\ -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu-1} \end{pmatrix} \tag{2.23}$$

At this point, since the locations of the errors are known. The error-locations, X_l^j, are substituted in (2.15) which can be expressed as

$$\begin{pmatrix} X_1^1 & X_2^1 & X_3^1 & \dots & X_\nu^1 \\ X_1^2 & X_2^2 & X_3^2 & \dots & X_\nu^2 \\ X_1^3 & X_2^3 & X_3^3 & \dots & X_\nu^3 \\ \vdots & \vdots & \vdots & & \vdots \\ X_1^\nu & X_2^\nu & X_3^\nu & \dots & X_\nu^\nu \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_\nu \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_\nu \end{pmatrix} \tag{2.24}$$

$$\text{Let N be } \begin{pmatrix} X_1^1 & X_2^1 & X_3^1 & \dots & X_\nu^1 \\ X_1^2 & X_2^2 & X_3^2 & \dots & X_\nu^2 \\ X_1^3 & X_2^3 & X_3^3 & \dots & X_\nu^3 \\ \vdots & \vdots & \vdots & & \vdots \\ X_1^\nu & X_2^\nu & X_3^\nu & \dots & X_\nu^\nu \end{pmatrix} \tag{2.25}$$

The error-magnitudes at specific location are determined as

$$\begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_\nu \end{pmatrix} = N^{-1} \begin{pmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_\nu \end{pmatrix} \tag{2.26}$$

Both the location and the magnitude of the errors have been resolved. The erroneous message can now be corrected to obtain transmitted message.

2.3.2 Example: Decoding RS(7,3) using GZP Algorithm

After transmitting the codeword(2.6) through the noisy channel, let the received data be

$$r(x) = \alpha^2 x^5 + \alpha^3 x^2 + \alpha^5 x \tag{2.27}$$

Calculating the error pattern, $S_j, 0 \leq j \leq 3$

$$\begin{aligned} S_0 &= r(\alpha^0) = \alpha^2 + \alpha^3 + \alpha^5 = 0 \\ S_1 &= r(\alpha^1) = \alpha^7 + \alpha^5 + \alpha^6 = \alpha^3 \\ S_2 &= r(\alpha^2) = \alpha^{12} + \alpha^7 + \alpha^7 = \alpha^5 \\ S_3 &= r(\alpha^3) = \alpha^{17} + \alpha^9 + \alpha^8 = \alpha^6 \end{aligned}$$

The syndrome polynomial is then represented as

$$S(x) = \alpha^6 x^3 + \alpha^5 x^2 + \alpha^3 x \tag{2.28}$$

The next step is to derive the error-locator polynomial leading to the error locations. With respect to (2.21), ν is two and therefore:

$$\begin{pmatrix} S_0 & S_1 \\ S_1 & S_2 \end{pmatrix} \begin{pmatrix} \lambda_2 \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} -S_2 \\ -S_3 \end{pmatrix} \tag{2.29}$$

$$\text{where M} = \begin{pmatrix} S_0 & S_1 \\ S_1 & S_2 \end{pmatrix} = \begin{pmatrix} 0 & \alpha^3 \\ \alpha^3 & \alpha^5 \end{pmatrix}$$

$$\begin{aligned} \det(\text{M}) &= \left| \begin{pmatrix} 0 & \alpha^3 \\ \alpha^3 & \alpha^5 \end{pmatrix} \right| \\ &= 0 + \alpha^6 = \alpha^6 \end{aligned} \tag{2.30}$$

Since det(M) $\neq 0$, the number of error $e = 2$.

The error-locator polynomial is then represented as $\Lambda(x) = \lambda_2 x^2 + \lambda_1 x + 1$, where λ_2 and λ_1 are then calculated as:

$$\begin{aligned} \begin{pmatrix} \lambda_2 \\ \lambda_1 \end{pmatrix} &= M^{-1} \begin{pmatrix} S_2 \\ S_3 \end{pmatrix} \\ &= \frac{\begin{pmatrix} S_2 & S_1 \\ S_1 & S_0 \end{pmatrix}}{|M|} \begin{pmatrix} S_2 \\ S_3 \end{pmatrix} \\ &= \frac{\begin{pmatrix} \alpha^5 & \alpha^3 \\ \alpha^3 & 0 \end{pmatrix} \begin{pmatrix} \alpha^5 \\ \alpha^6 \end{pmatrix}}{\alpha^6} \\ &= \begin{pmatrix} \frac{\alpha^{10}+\alpha^9}{\alpha^6} \\ \frac{\alpha^8}{\alpha^6} \end{pmatrix} = \begin{pmatrix} \alpha^6 \\ \alpha^2 \end{pmatrix} \end{aligned} \tag{2.31}$$

The error locator polynomial is $\Lambda(x) = \alpha^6 x^2 + \alpha^2 x + 1$

The Chien's algorithm is used to calculate the location of the errors from the error-locator polynomial, $\Lambda(x)$

$$\Lambda(\alpha^0) = \alpha^6 + \alpha^2 + 1 = 0 \tag{2.32}$$

$$\Lambda(\alpha^1) = \alpha^8 + \alpha^3 + 1 = 0 \tag{2.33}$$

$$\Lambda(\alpha^2) = \alpha^{10} + \alpha^4 + 1 = \alpha^2$$

$$\Lambda(\alpha^3) = \alpha^{12} + \alpha^5 + 1 = 1$$

$$\begin{aligned}
\Lambda(\alpha^4) &= \alpha^{14} + \alpha^6 + 1 = \alpha^6 \\
\Lambda(\alpha^5) &= \alpha^{16} + \alpha^7 + 1 = \alpha^2 \\
\Lambda(\alpha^6) &= \alpha^{18} + \alpha^8 + 1 = \alpha^6
\end{aligned}$$

The zeros of $\Lambda(x)$ are X_1 and X_2 which are equivalent to α^{-0} and α^{-1} respectively. According to (2.17), the location of the errors given as position 0 and 6. The error-locations are then substituted in the the syndrome set of equation (2.28) giving

$$\begin{aligned}
S_1 &= r(\alpha^1) = e_6\alpha^6 + e_0 = \alpha^3 \\
S_2 &= r(\alpha^2) = e_6\alpha^{12} + e_0 = \alpha^5
\end{aligned}$$

The above simultaneous equation can be expressed in matrix form

$$\begin{pmatrix} \alpha^6 & 1 \\ \alpha^5 & 1 \end{pmatrix} \begin{pmatrix} e_6 \\ e_0 \end{pmatrix} = \begin{pmatrix} \alpha^3 \\ \alpha^5 \end{pmatrix}$$

$$\begin{aligned}
\text{Therefore, } \begin{pmatrix} e_6 \\ e_0 \end{pmatrix} &= \begin{pmatrix} \alpha^6 & 1 \\ \alpha^5 & 1 \end{pmatrix}^{-1} \begin{pmatrix} \alpha^3 \\ \alpha^5 \end{pmatrix} \\
&= \frac{\begin{pmatrix} 1 & 1 \\ \alpha^5 & \alpha^6 \end{pmatrix} \begin{pmatrix} \alpha^3 \\ \alpha^5 \end{pmatrix}}{\alpha} \\
&= \begin{pmatrix} \frac{\alpha^3+\alpha^5}{\alpha} \\ \frac{\alpha^8+\alpha^{11}}{\alpha} \end{pmatrix} = \begin{pmatrix} \alpha \\ \alpha \end{pmatrix}
\end{aligned} \tag{2.34}$$

The error polynomial is given by $e(x) = \alpha x^6 + \alpha$ and the corrected message is:

$$\begin{aligned}
c(x) &= r(x) + e(x) \\
&= \alpha^2 x^5 + \alpha^3 x^2 + \alpha^5 x + \alpha x^6 + \alpha \\
&= \alpha x^6 + \alpha^2 x^5 + \alpha^3 x^2 + \alpha^5 x + \alpha
\end{aligned} \tag{2.35}$$

The GPZ algorithm provides the basis of the Reed-Solomon decoder. However, this algorithm is limited in the number of errors it can correct before the implementation becomes

too computational intensive due to the matrix inversion. To efficiently correct large number of errors, other approaches have been developed and among those are the Berlekamp-Massey and the euclidean algorithm.

2.3.3 Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm was first introduced in 1965. It is the first efficient algorithm to decode the RS code. The Berlekamp and Massey's approach is to generate a Linear Feedback Shift Register (LFSR) of minimal length and with the required feedback λ such that the first $2t$ elements in the LFSR output sequence shall correspond to the syndromes S_1, $S_2, ...,$ S_{2t} to satisfy (2.20). The taps of the LFSR represents the coefficients of (2.16), which uniquely specifies the error-locator polynomial.

Figure 2.4 shows the flowchart of the Berlekamp-Massey Algorithm. Once the Berlekamp-Massey LFSR has been constructed, the feedback taps represent the coefficients of the error-locator polynomial. Having the error-locator polynomial $\Lambda(x)$ and the Syndrome polynomial $S(x)$, the error-evaluator polynomial $\Omega(x)$ is solved from the key equation (2.36).

$$\Lambda(x)S(x) \equiv \Omega(x) \; mod \; x^{2t+1} \tag{2.36}$$

Once the error-locator polynomial and the error-evaluator polynomial have been derived, the error locations and error magnitudes are solved using the Chien's [6] and Forney's [11] algorithm, respectively.

2.3.4 Example: Decoding RS(7,3) using BM Algorithm

The same set of data as in Example 2.3.2 is used. The received polynomial is given as

$$r(x) = \alpha^2 x^6 + \alpha^2 x^5 + \alpha^5 x + \alpha \tag{2.37}$$

and the previously calculated syndrome is expressed as

$$S(x) = \alpha^4 x^3 + \alpha^6 x^2 + \alpha^2 x + \alpha^6 \tag{2.38}$$

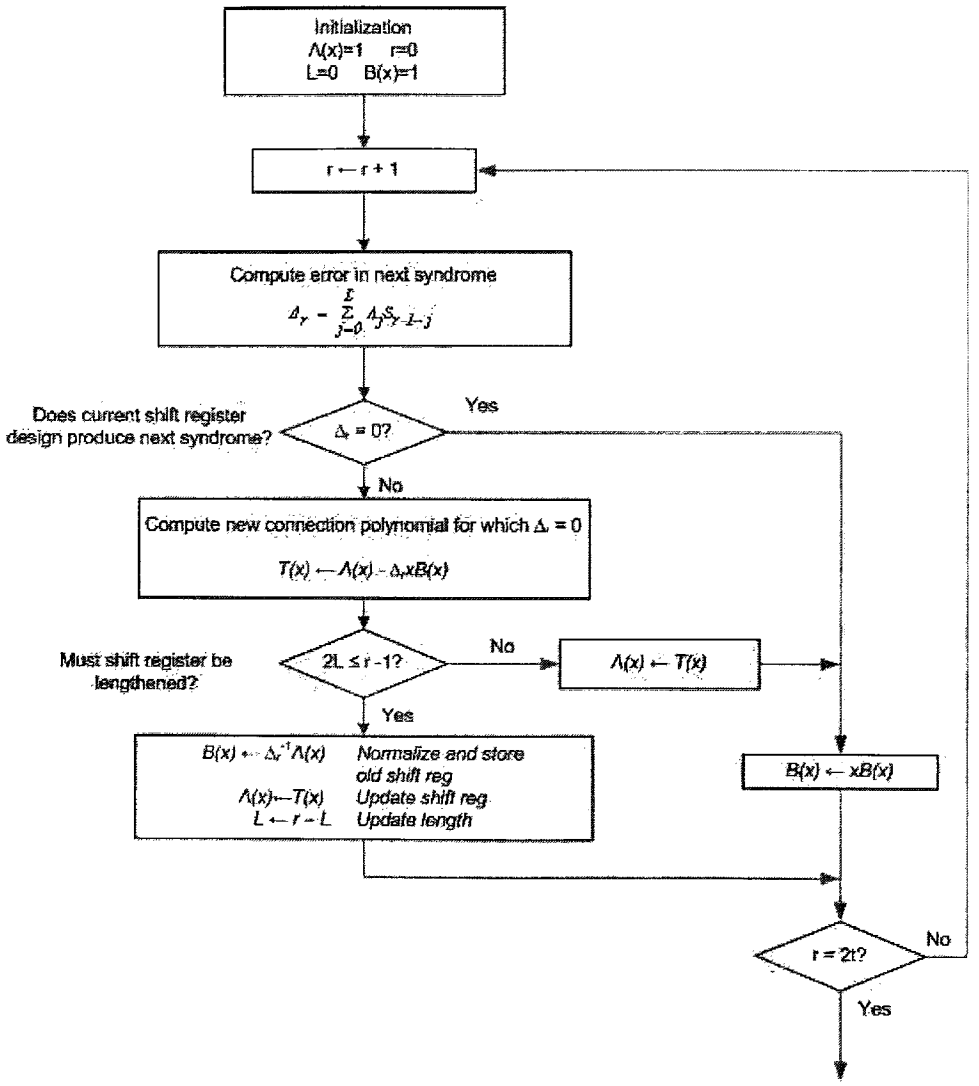

Figure 2.4: Berlekamp-Massey Algorithm [4]

The error-locator polynomial is calculated using the flowchart in Figure 2.4. Table 2.3 shows the intermediate values of different variables at different iteration. The number of iterations in this example is four because the RS code can correct up to two errors.

r	Δ_r	T(x)	B(x)	$\Lambda(x)$	L
0			1	1	0
1	α^6	$1+\alpha^6 x$	α	$1+\alpha^6 x$	1
2	α^3	$1+\alpha^3 x$	$\alpha^4+\alpha^3 x$	$1+\alpha^3 x$	1
3	α	$1+\alpha^2 x+\alpha^4 x^2$	$\alpha^6 x^2$	$1+\alpha^2 x+\alpha^4 x^2$	2
4	1	$1+x+\alpha x^2$	$\alpha^6+\alpha^2 x^2$	$1+x+\alpha x^2$	

Table 2.3: Berlekamp-Massey Table

The resulting error-locator polynomial is $\Lambda(x) = 1 + x + \alpha x^2$. The error evaluator polynomial is then calculated from the key equation:

$$\begin{aligned} \Omega(x) &= \Lambda(x)S(x) \bmod \mathrm{x}^{2t} \\ &= (\alpha x^2 + x + 1)(\alpha^4 x^3 + \alpha^6 x^2 + \alpha^2 x + \alpha^6) \bmod \mathrm{x}^4 \\ &= x + \alpha^6 \end{aligned} \tag{2.39}$$

Using the Chien's algorithm, the locations of the errors are searched

$$\begin{aligned} \Lambda(\alpha^0) &= 1+\alpha^0+\alpha^1 = 1 \\ \Lambda(\alpha^1) &= 1+\alpha^1+\alpha^3 = 0 \\ \Lambda(\alpha^2) &= 1+\alpha^2+\alpha^5 = \alpha^1 \\ \Lambda(\alpha^3) &= 1+\alpha^3+\alpha^7 = \alpha^3 \\ \Lambda(\alpha^4) &= 1+\alpha^4+\alpha^9 = \alpha^3 \\ \Lambda(\alpha^5) &= 1+\alpha^5+\alpha^{11} = 0 \\ \Lambda(\alpha^6) &= 1+\alpha^6+\alpha^{13} = 1 \end{aligned} \tag{2.40, 2.41}$$

The zeros of $\Lambda(x)$ are X_1 and X_5, which are equivalent to α^{-1} and α^{-5} respectively. The location of the errors given as position 6 and 2.

The error polynomial is then calculated using the Forney's algorithm given by

$$\begin{aligned} e_l &= \frac{\Omega(X_l^{-1})}{X_l^{-1}\Lambda'(X_l^{-1})} \\ &= \frac{X_l^{-1} + \alpha^6}{X_l^{-1}1} \\ &= 1 + \alpha^6 X_l \end{aligned} \tag{2.42}$$

$$e_2 = 1 + \alpha^6\alpha^2 = \alpha^3 \tag{2.43}$$

$$e_6 = 1 + \alpha^6\alpha^6 = \alpha^4 \tag{2.44}$$

The error polynomial is $e(x) = \alpha^4x^6 + \alpha^3x^2$ and the recovered codeword $c(x) = \alpha x^6 + \alpha^2x^5 + \alpha^3x^2 + \alpha^5x + \alpha$ which is same to the transmitted codeword in (2.6). An another algorithm that could be used to decode the Reed-Solomon codes is the Euclidean Algorithm.

2.3.5 Euclidean Algorithm

The Euclidean Algorithm is known to be one of the oldest algorithm to calculate the greatest common denominator of two numbers. It is also applied to solve the Key Equation (2.36) as illustrated in Figure 2.5.

2.3.6 Example: Decoding RS(7,3) using Euclidean Algorithm

The same set of data as in Example 2.3.2 is used. The received polynomial is given as

$$r(x) = \alpha^2x^6 + \alpha^2x^5 + \alpha^5x + \alpha \tag{2.45}$$

and the syndrome is similar to previously calculated syndrome and is expressed as

$$S(x) = \alpha^4x^3 + \alpha^6x^2 + \alpha^2x + \alpha^6 \tag{2.46}$$

The error-locator and error-evaluator polynomials are calculated using the flowchart in Figure 2.5. Table 2.4 shows the intermediate values of different variables at different

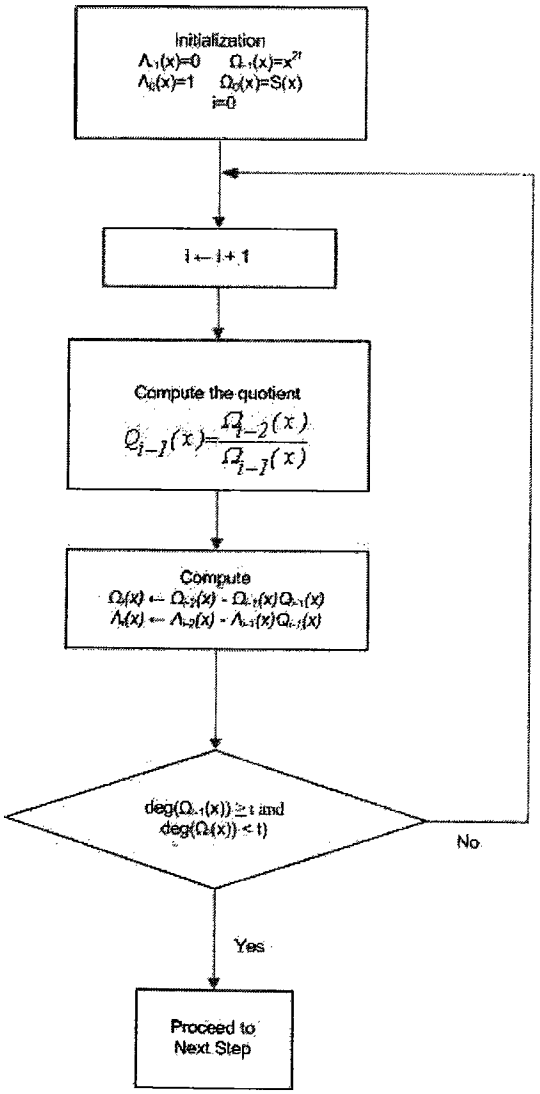

Figure 2.5: Euclidean Algorithm

iteration. The number of iterations in this example is two because the RS code can correct up to two errors.

The key equation is represented as

$$S(x)\Lambda(x) = \Omega(x) \bmod x^{2t} \tag{2.47}$$

$$S(x)B_i(x) = R_i(x) \bmod x^{2t} \tag{2.48}$$

i	$R_i(\mathbf{x})$	$Q_i(\mathbf{x})$	$B_i(\mathbf{x})$
-1	x^4	-	0
0	$\alpha^4x^3+\alpha^6x^2+\alpha^2x+\alpha^6$	$\alpha^3x+\alpha^5$	1
1	$x^2+\alpha^6x+\alpha^4$	$\alpha^4x+\alpha^4$	$\alpha^3x+\alpha^5$
2	$\alpha^6x+\alpha^5$		$x^2+\alpha^6x+\alpha^6$

Table 2.4: Euclidean's Algorithm Table

From Table 2.4, the error-locator polynomial is given as $\Lambda(x) = x^2 + \alpha^6x + \alpha^6$ the error-evaluator polynomial is given as $\Omega(x) = \alpha^6x + \alpha^5$.

Using the Chien's algorithm to calculate the location of the error

$$\begin{aligned}
\Lambda(\alpha^0) &= 1+\alpha^0+\alpha^1 = 1 \\
\Lambda(\alpha^1) &= 1+\alpha^1+\alpha^3 = 0 \\
\Lambda(\alpha^2) &= 1+\alpha^2+\alpha^5 = \alpha^1 \\
\Lambda(\alpha^3) &= 1+\alpha^3+\alpha^7 = \alpha^3 \\
\Lambda(\alpha^4) &= 1+\alpha^4+\alpha^9 = \alpha^3 \\
\Lambda(\alpha^5) &= 1+\alpha^5+\alpha^{11} = 0 \\
\Lambda(\alpha^6) &= 1+\alpha^6+\alpha^{13} = 1
\end{aligned} \tag{2.49, 2.50}$$

The zeros of $\Lambda(x)$ are X_1 and X_5 which are equivalent to α^{-1} and α^{-5} respectively. The location of the errors given as position 6 and 2. The error polynomial is then calculated using the Forney's algorithm given by

$$e_l = \frac{\Omega(X_l^{-1})}{X_l^{-1}\Lambda'(X_l^{-1})} \tag{2.51}$$

Architecture	Adders	Multipliers	Latches	Muxes	Clock Cycles
iBM (Blahut)	2t+1	3t+3	4t+2	t+1	3t
iBM (Berlekamp)	3t+1	5t+3	6t+2	2t+1	2t
riBM	2t+1	3t+3	4t+2	t+1	2t
RiBM	2t+1	3t+3	4t+2	t+1	3t
Euclidean	2t+1	3t+3	4t+2	t+1	3t
Euclidean	2t+1	3t+3	4t+2	t+1	3t

Table 2.5: Comparison of Hardware Complexity and Path Delays [31]

where $\Lambda'(x) = \frac{\Lambda_{odd}(x)}{x} = \alpha^6$

Therefore,

$$e_l = \frac{\Omega(X_l^{-1})}{X_l^{-1}\Lambda'(X_l^{-1})} = \frac{\alpha^6 X_l^{-1} + \alpha^5}{X_l^{-1}\alpha^6} = 1 + \alpha^6 X_l \tag{2.52}$$

$$e_2 = 1 + \alpha^6\alpha^2 = \alpha^3 \tag{2.53}$$

$$e_6 = 1 + \alpha^6\alpha^6 = \alpha^4 \tag{2.54}$$

The error polynomial is $e(x) = \alpha^4 x^6 + \alpha^3 x^2$ and the recovered codeword $c(x) = \alpha x^6 + \alpha^2 x^5 + \alpha^3 x^2 + \alpha^5 x + \alpha$ which is same to the transmitted codeword in (2.6). Both the Euclidean and the BM algorithm can be used to solve the key equation.

2.4 Implementation Issues

Many ASIC implementations of the Reed-Solomon codec have been described in literature. Two most common algorithms used to solve the "Key Equation" are the Euclidean [18] and the Berlekamp-Massey [4] algorithms. On one hand, the Euclidean algorithm has been argued to be easy to implement, but requires much resources because of the long division. On the other hand, the Berlekamp-Massey algorithm has high complexity level but is area-efficient [31]. Table 2.5 summerizes the complexity of different architectures described in [31].

It is observed that the derivatives of the Berlekamp algorithm outperformed the derivative of Euclidean algorithm in ASICs. However, this might not be necessarily true on

FPGA since each FPGA family has their own architecture. Part of this research is geared toward how the FPGA architecture impact the performance of a specific RS codec. Since the Berlekamp algorithm is claimed to be area-efficient, it is the chosen to be implemented as the Key Equation Solver in the RS codec.

Chapter 3

Design & Implementation

The designed system consists of an encoder and a decoder as illustrated in Figure 3.1. Before transmitting a message $m(x)$, the encoder appends $2t$ parity symbols to k message symbols to form a codeword $c(x)$ of n symbols. Upon receiving the codeword $r(x)$, the decoder detects and evaluates $e(x)$, which is the error polynomial introduced during transmission over the noisy channel. The XOR operation is then performed on $r(x)$ and $e(x)$ to recover the transmitted codeword, $c(x)$. The message polynomial, $m(x)$, is represented by the first k symbols of $c(x)$.

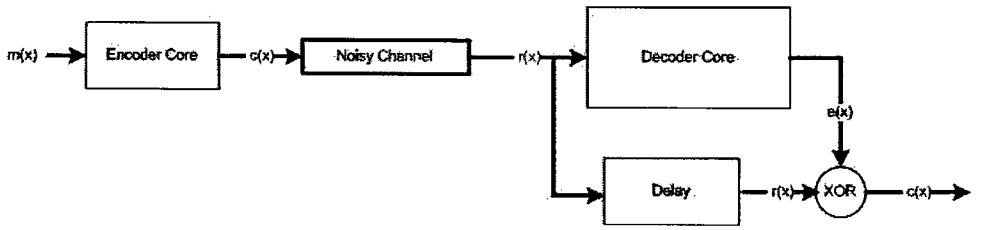

Figure 3.1: System Block Diagram

In the OTN G.709 specification, the RS code, RS(255,239), is used for FEC. One symbol contains 8 bits and the length of the codeword is 255 bytes. RS(255,239) contains 239 data symbols and can correct up to a maximum of 8 erroneous bytes. The following sections go over the architectural design for implementing the algorithms described in the Chapter2.

3.1 Encoder

The encoder reads the message to be transmitted, calculates and appends the parity bytes at the end of the message. A block diagram of the encoder is shown in Figure 3.2. The *i_start* signal indicates the beginning of each message to be encoded. After the 239^{th} byte has been pushed into the encoder, the *i_enable* signal is asserted to shift the parity bytes out of the register.

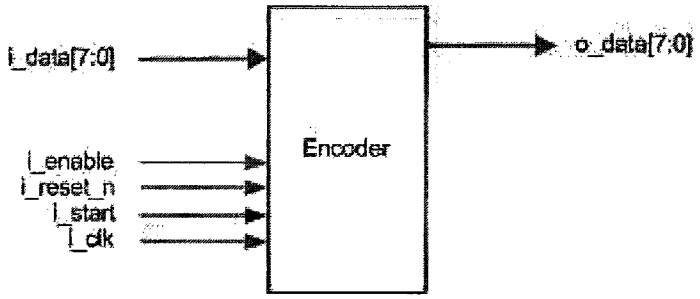

Figure 3.2: Encoder Block Diagram

The codeword, $c(x)$ is systematically encoded using (3.1).

$$c(x) = m(x)X^{16} + m(x)X^{16} \bmod g(x) \tag{3.1}$$

The systematic encoder is implemented using a Linear Feedback Shift Register (LFSR) as shown in Figure 3.3. At the end of 239 clock cycles, the contents in LFSR are appended to the 239 bytes of data.

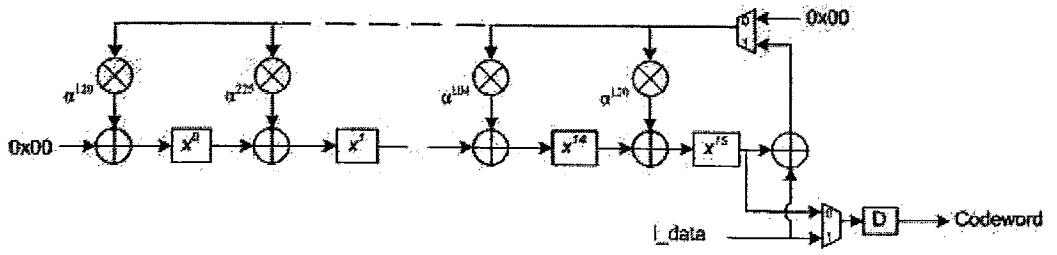

Figure 3.3: Encoder Architecture

The taps of the LFSR are defined by the coefficients of the generator polynomial $g(x)$

of degree 16, which is given by

$$g(x) = \prod_{i=0}^{15} (x + \alpha^i) \tag{3.2}$$

Using the Galois field based on the primitive polynomial, $p(z) = z^8 + z^4 + z^3 + z^2 + 1$, the coefficients of the generator polynomial, $g(x)$ are derived as

$$\begin{aligned} g(x) &= x^{16} + \alpha^{120}x^{15} + \alpha^{104}x^{14} + \alpha^{107}x^{13} + \alpha^{109}x^{12} + \alpha^{102}x^{11} \\ &+ \alpha^{161}x^{10} + \alpha^{76}x^9 + \alpha^3x^8 + \alpha^{91}x^7 + \alpha^{191}x^6 + \alpha^{147}x^5 \\ &+ \alpha^{169}x^4 + \alpha^{182}x^3 + \alpha^{194}x^2 + \alpha^{225}x + \alpha^{120} \end{aligned} \tag{3.3}$$

3.2 Decoder

The top-level block diagram of the decoder is shown in Figure 3.4. The *i_start* signal is pulsed every 255 clock cycles to indicate the beginning of each received codeword to be decoded and the output of a syndrome polynomial from the syndrome calculator. The Key Equation Solver will then take 224 clock cycles to generate the error-locator polynomial and the error-evaluator polynomial from the syndrome polynomial. Once these two polynomials are formed, the Chien's algorithm [6] uses the error-locator polynomial to exhaustively search for the locations of the errors. After the errors have been located, the Forney's algorithm [11] will calculate the magnitude of the errors from the error-evaluator polynomial. The decoder validates the codeword on the last byte of the codeword. If the received

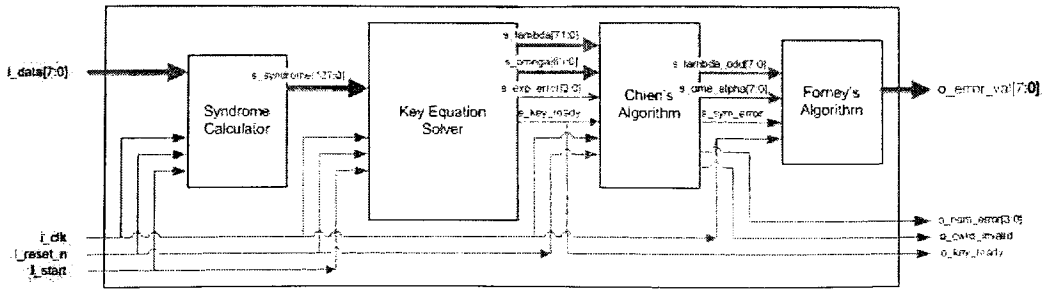

Figure 3.4: Decoder Block Diagram

codeword contains more than eight errors, the uncorrectable codeword is discarded. Otherwise, the decoder will generate the error polynomial, $e(x)$ through *o_error_val*. As the decoder outputs the error value, the *o_key_ready* signal is used to fetch the corresponding received symbol from the FIFO, used to buffer the received symbols during the decoding process. The decoder also outputs the number of corrected errors through *o_num_errors* for statistical records.

3.2.1 Syndrome Calculator

Figure 3.5 shows the block diagram of the syndrome calculator, which is used to detect errors in the RS decoding scheme. After the *i_start* signal has been pulsed, the received

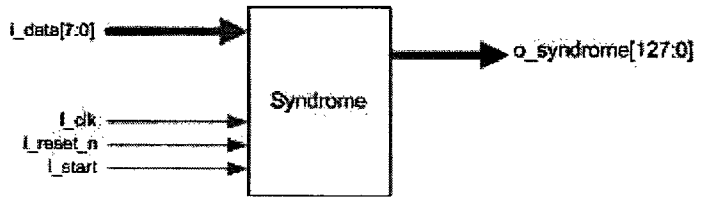

Figure 3.5: Syndrome Calculator Block Diagram

codeword is fed into the syndrome calculator through *i_data* at a rate of one byte per clock. It takes 255 clock cycles to calculate the syndrome polynomial. In the next clock cycle, the contents of the syndrome registers are shifted into the Key Equation Solver in parallel through *s_syndrome* which is a 16 x 8 bit-signal. In the same clock cycle, the syndrome registers are preset with the first symbol of the next codeword.

The Syndrome Calculator implements the function S(x) given in (2.9). As each byte gets in the syndrome calculator block, it is multiplied by all the roots of the generator polynomial, $g(x)$, in parallel as shown in Figure 3.6. Recalling $g(x)$ from (2.4), the roots of $g(x)$ are given as α^i, where $0 \leq i \leq 15$. At the end of 255 clock cycles, after the last byte has been processed, the 16 accumulator registers contain the coefficients, $r(\alpha^0)$, $r(\alpha^1)$, ..., $r(\alpha^{15})$ of the syndrome polynomials, $S(x)$. These coefficients are stored in a 128 bit register, which will be used by the Key Equation Solver. The latency of the

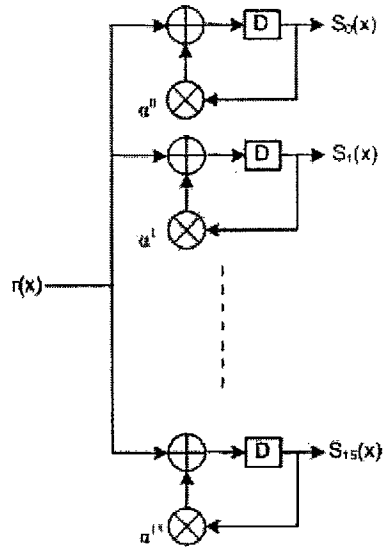

Figure 3.6: Syndrome Calculator Architecture

syndrome calculator is 255 clock cycles.

3.2.2 Key Equation Solver

The Key Equation Solver block is the most complex block to implement. The block diagram of this Key Equation Solver is shown in Figure 3.7. When the syndrome coefficients are ready to be processed, the *i_start* signal is triggered to shift the coefficients into the Key Equation Solver. Once this signal is pulsed, the Key Equation Solver block will

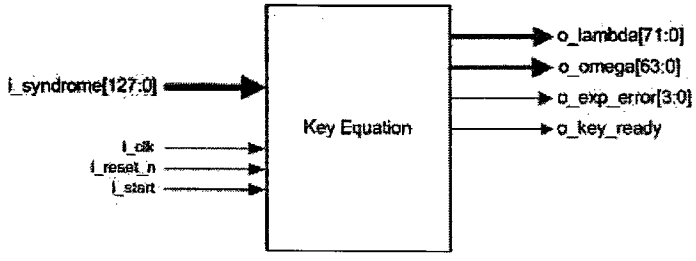

Figure 3.7: Key Equation Solver Block Diagram

take 224 clock cycles to generate the coefficients of error-locator polynomial, $\Lambda(x)$, and

error-evaluator polynomials, $\Omega(x)$. The *o_exp_error* signal represents the number of errors estimated by the Key Equation Solver and is used by the Error-Locator block to validate the codeword.

The Berlekamp-Massey algorithm from Figure 2.4 is translated into a state machine shown in Figure 3.8. The state machine is started by a pulse from *i_start* signal. The register $\Lambda(x)$ is initialized to one and the expected number of errors L is set to zero. The

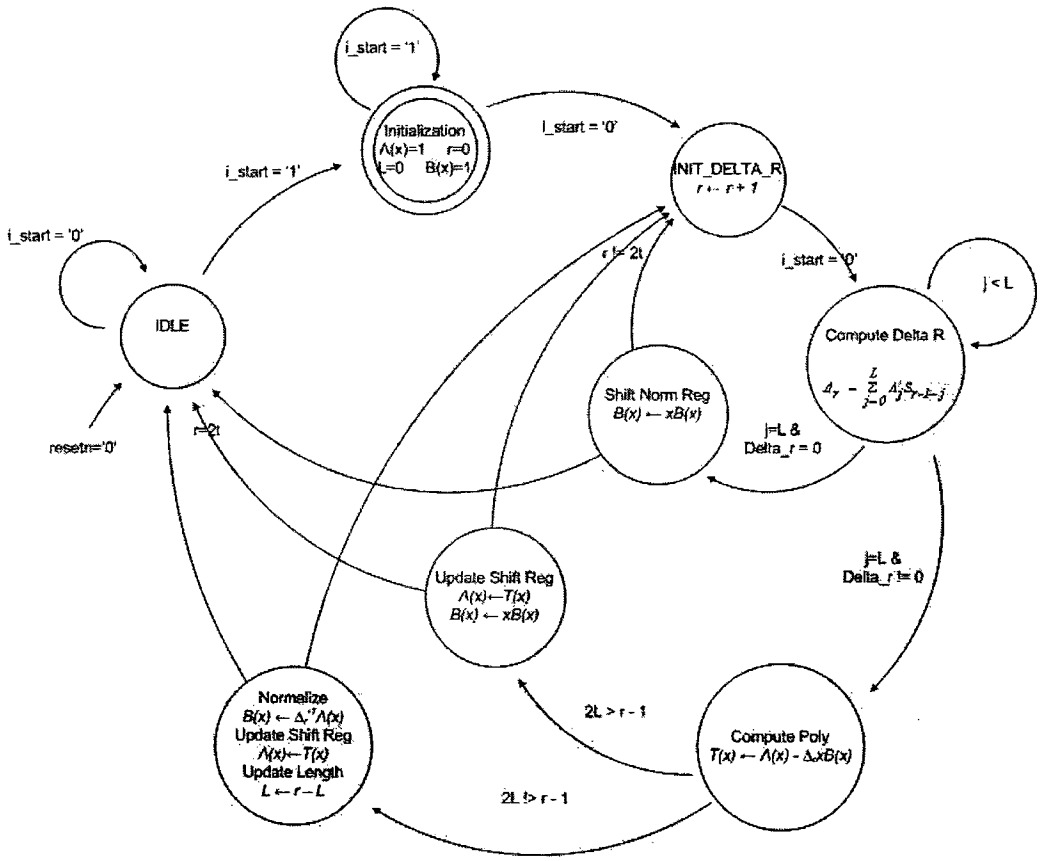

Figure 3.8: Berlekamp-Massey State Diagram

state machine stops when r equals to $2t$. At this point, the register $\Lambda(x)$ contains the coefficients of the error-locator polynomial. The next step is to calculate the coefficients of the error-evaluator polynomial $\Omega(x)$. This is done by solving the Key Equation shown in (2.36) for $\Omega(x)$. This version of *Key Equation Solver* takes 140 clock cycles, which

is less than 224 clock cycles for the final design, to generate $\Lambda(x)$ and $\Omega(x)$. However, this implementation causes the overall system to run at a lower clock rate, and requires improvement.

After analyzing the state machine, a derivative of the above implementation is proposed to speed up the design and optimize for resource utilization. The first improvement made is to eliminate the register, $T(x)$, which is internal to a loop of the state machine. Eliminating $T(x)$ allows to combine the *Shift Norm Reg* and *Update Shift Reg* states in Figure 3.8. To compensate for $T(x)$, the normalizing process ($B(x) \leftarrow \Delta_r^{-1}\Lambda(x)$) is moved to state *Compute Poly* as shown in Figure 3.9. The shift register is normalized only if the condition $2L \leq r - 1$ has been satisfied. When normalizing the shift register, the arithmetic,

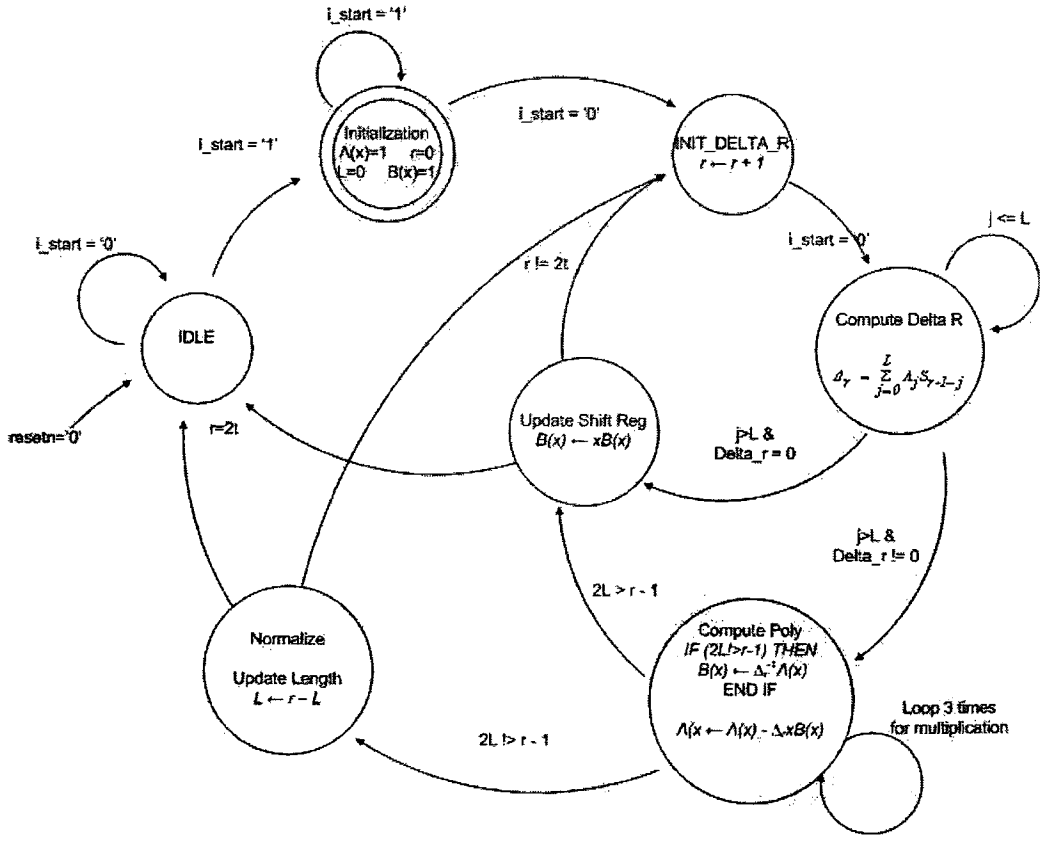

Figure 3.9: Proposed Berlekamp-Massey State Diagram

$\triangle_r^{-1}\Lambda(x)$, is evaluated. In the *Compute Poly* state, $\triangle_r^{-1}$ represents the finite field inverse of $\triangle_r$. A typical logic implementation [14] of finite field inversion in $GF(2^8)$ usually consumes 16 clock cycles before generating an answer. Unfortunately, in this state machine design, 16 clock cycles for one inversion is unacceptable. Before generating the locator polynomial, $\Lambda(x)$, there would be a most 8 inversions, which would consume 128 clock cycles. So, the state machine would need 260 clock cycles. But, the Key Equation Solver has only 255 clock cycles to generate $\Lambda(x)$ before receiving the next syndrome polynomial, $S(x)$. Instead of using logics to implement the finite field inverse, a look-up table with 255 entries is used. Each entry of the look-up table represents the address of the location where the inverse of an finite field element is stored.

The *Compute_Poly* state is now responsible for two operations: $B(x) \leftarrow \Delta_r^{-1}\Lambda(x)$ and $\Lambda(x) \leftarrow \Lambda(x) - \Delta_r x B(x)$. These two operations are executed in parallel and consumes a total of eighteen finite field multipliers. The proposed implementation in Figure 3.9 serializes both functions at the expense of latency. In the new design, each function shares three multipliers to multiply different coefficients at different clock. As a result, the number of Galois Field multipliers is reduced from eighteen down to six. However, the cost of resource sharing is latency. The latency of the Key Equation Solver has been increased from 140 cycles to 224 cycles. An increase in latency leads to an increase in the FIFO depth, used to buffer the received symbols during the decoding process. However, the FIFO can be implemented in block RAMs, which are available in both Virtex4 and StratixII. These block RAMs are not part of the programmable logics and they are wasted if they are not used.

The speed of the initial implementation was about half the targeted clock rate. When analyzing the timing report, most critical path contains multiplexers instantiated by the synthesis tools. A pipeline register is added in the critical path to decrease the delay. The state machine is then modified to compensate the additional pipeline registers. The use of the pipeline registers helps the decoder to achieve the speed specified by the OTN G.709 [20] and the resource sharing improves the resource utilization by 15%.

3.2.3 Error Locator

The objective of this Error Locator block, as shown in Figure 3.10, is to generate a error polynomial with identified locations of the errors. The error-locator takes in the error-locator polynomial, $\Lambda(x)$, and error-evaluator polynomial, $\Omega(x)$, through *i_lambda* and *i_omega* from the Key Equation Solver, respectively. When the *i_key_ready* signal is pulsed, the error-locator starts searching for the locations of the errors by evaluating the error-locator polynomial, $\Lambda(\alpha^i)$. The index i ranges from 0 to 254 and is incremented by unity for every clock cycle. If $\Lambda(\alpha^i) = 0$, the symbol at location $(255 - i)$ is erroneous and is marked by *o_sym_error* signal. Such algorithm is known as the Chien's algorithm [6].

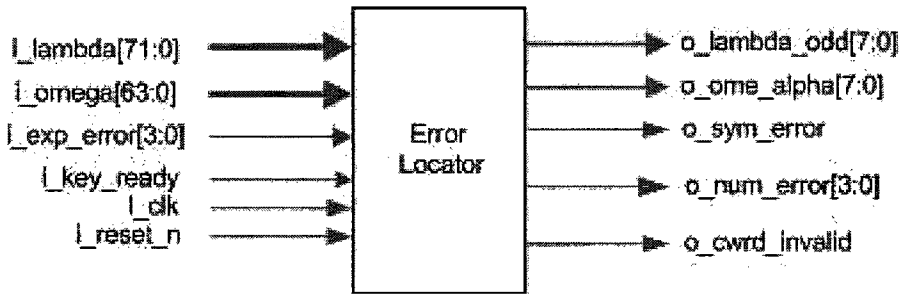

Figure 3.10: Error Locator Block Diagram

The architecture to implement the Chien's algorithm is shown on the left of Figure 3.11. Once the error-location polynomial, $\Lambda(x)$, has been generated, its coefficients λ_0, λ_1, ..., λ_8 of the error-locator polynomials are preset in the registers as shown in Figure 3.11. The output at this operation is equivalent to $\Lambda(\alpha^0)$. On the following clock cycles, the values in the registers are multiplied by corresponding *alpha* to evaluate $\Lambda(\alpha^1)$, $\Lambda(\alpha^2)$,...$\Lambda(\alpha^{254})$. If $\Lambda(\alpha^i) = 0$, then the $(255 - i)^{th}$ symbol of the received polynomial is erroneous and, therefore, its error magnitude needs to be evaluated. For each codeword, *o_sym_error* flags the locations of the erroneous symbols. Each time the error-locator finds an error, it will increment a counter to monitor the number of errors. If the number of errors obtained by the error-locator does not match with the number of errors calculated by the key equation solver, the codeword is marked as invalid and is discarded. However, if the number

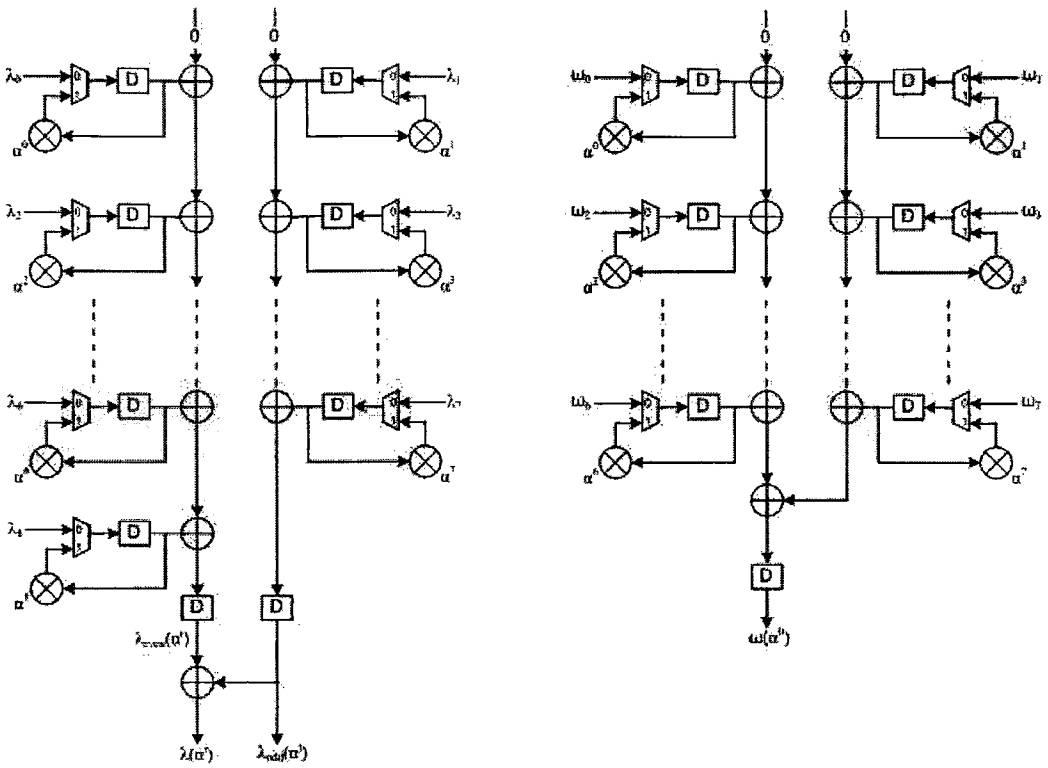

Figure 3.11: Error Locator Architecture

matches, the transmitted codeword can be recovered.

3.2.4 Error Evaluator

The error evaluator block, as shown in Figure 3.12, is the last step in decoding RS codes. The error-evaluator is based on the Forney's algorithm [11], and calculates the error values that have been introduced during the transmission. The *i_sym_error* signal represents the

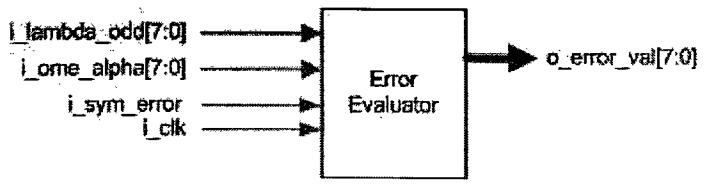

Figure 3.12: Error Evaluator Block Diagram

locations of the symbols that need to be corrected. The error is calculated according to (3.4), which is derived from (2.51) since $\Lambda'(X_l) = X_l\Lambda_{odd}(X_l)$.

$$e_c = \frac{\Omega(X_l^{-1})}{\Lambda_{odd}(X_l^{-1})} \tag{3.4}$$

X_l represents the index i in error locating process and X_l^{-1} represents the location $255 - i$ in the receiving codeword. The implemented architecture is based on (3.4) to evaluate the error magnitude. The coefficients of $\Omega(X_l^{-1})$ and $\Lambda_{odd}(X_l^{-1})$ are calculated in the Error-Locator block as shown in Figure 3.11. No additional hardware was required to implement term $\Lambda_{odd}(X_l^{-1})$ because $\Lambda(X_l^{-1})$ was calculated as the sum of $\Lambda_{even}(X_l^{-1})$ and $\Lambda_{odd}(X_l^{-1})$. A Look-Up Table is used to store the inverse value of $\lambda_{odd}(\alpha^i)$ as shown in Figure 3.13. The look-up table is coded in VHDL such that it infers memories instead of logics since memories are readily available in both Virtex4 and StratixII. This is done by registering the addresses and the outputs of the look-up table. As a result, the logic can instead be used for other functional blocks and thus improving the FPGA logic utilization.

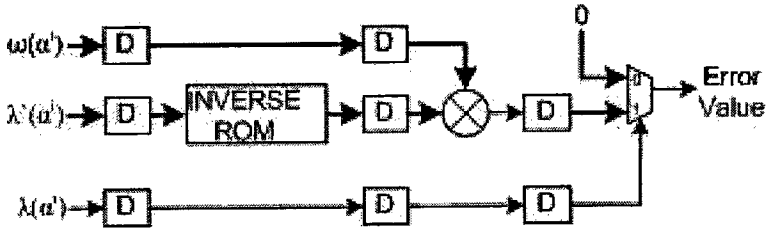

Figure 3.13: Error Evaluator Architecture

3.3 Summary

The proposed RS(255,239) codec is implemented using VHDL. The encoder has a latency of one clock cycle and the decoder has a latency of 488 clock cycles. The FIFO should at least be to able to store 488 bytes of incoming codewords. The overall challenge in implementing the RS codec is the Key Equation Solver block. A state machine approach was used to implement the Berlekamp-Massey algorithm. The original design was performing

at only half the speed required by OTU-3. So, the state machine was redesigned with additional pipeline stages to speed up the rate. The VHDL code for this block was written in a behavioral style and portable to either Altera's or Xilinx's FPGA devices. The other building blocks were implemented in an architectural coding style where the hardware resources are inferred. The implemented RS codec is simulated in Modelsim v6.0 and synthesized targeting Xilinx's Virtex4 and Altera's StratixII.

Chapter 4

Simulation & Synthesis Results

4.1 Verification

4.1.1 MATLAB Model

To verify the functionality of the implemented RS codec, a model of the system is built using simulink from MATLAB as shown in Figure 4.1. The Message block is used to generate a random sequence of 239 bytes, which are saved in the *message* file. The 239-byte message is encoded to a 255-byte codeword, which is stored in the *codeword* file. The *message* and *codeword* files are used as a baseline to test implemented codec in the verification environment. The file *r_x* is used to confirm that the receiving data are erroneous. Different *Initial Seed* are used to generate test files with different patterns.

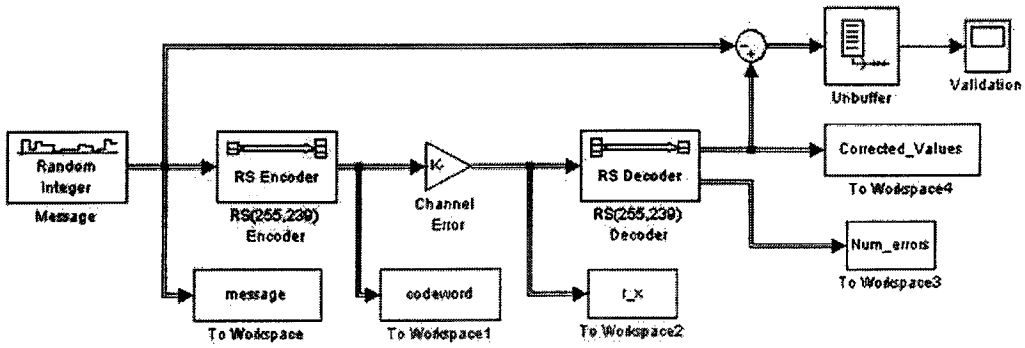

Figure 4.1: RS Model

The codeword is then corrupted and the erroneous codeword, *r_x*, is then decoded. In this model, the RS decoder outputs the number of corrected errors and the corrected codewords if the number of errors is eight or fewer. The decoded codeword is compared to the generated message and the resulting frame is converted to a scalar samples output at a higher sample rate. A horizontal line crossing the origin,(Figure 4.2 (left)), indicates

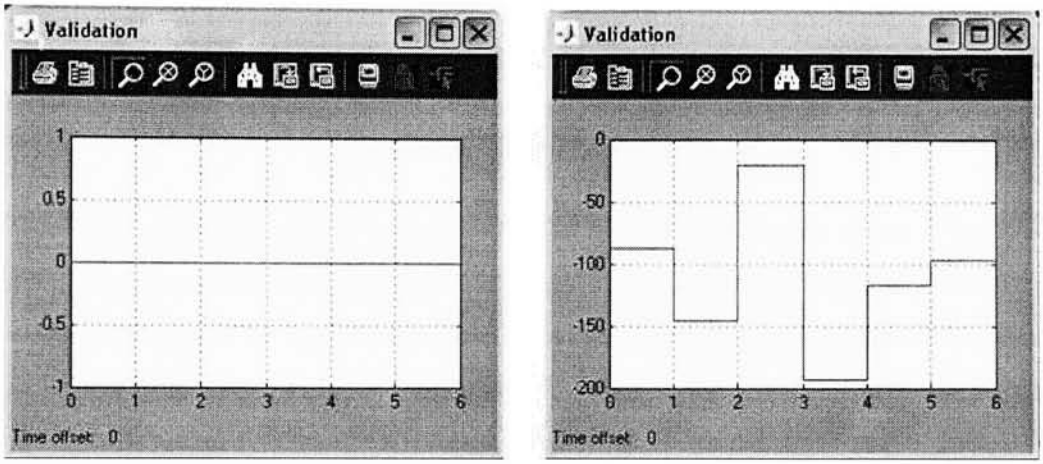

Figure 4.2: Validation of RS Codec model

that the decoded message is equal to the generated message. However if more than the maximum limit number of errors have been injected into the codeword, a step function-like signal will be observed on the scope as shown in Figure 4.2 (right). If the number of errors is more than eight, the RS(255,239) codeword is not correctable and must be discarded.

4.1.2 Error Generator

The RS(255,239) decoder can correct up to eight symbol errors. An error generator, shown in Figure 4.3, is used to produce random errors at random locations in a codeword. The Random Location block generates a random binary error pattern consisting of 0s and 1s, where a 1 indicates that an error will be injected in the corresponding location. A probability of 3.5% for the Random Location block will generate codeword with an average of 9 errors out of 255 symbol locations. The Random Magnitude block generates uniformly

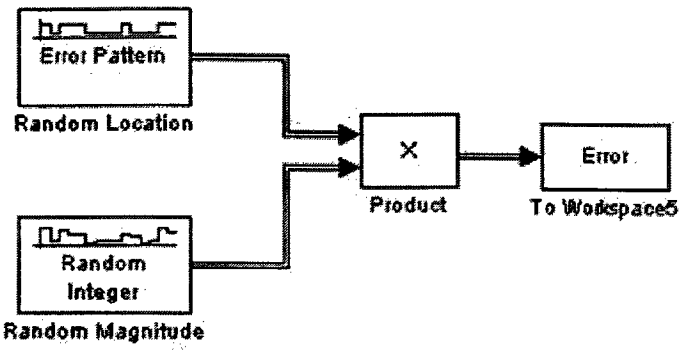

Figure 4.3: Error Generator

distributed random integers in the range of 1 and 255. Multiplying the output of the Random Location and Random Magnitude block produces a random error pattern to be added to the codeword. A set of sixteen error patterns are generated using different *Initial Seed* and stored in an error file. In the verification environment, the contents of the error file are added to the encoded codeword to corrupt the received codeword. The received codeword is then decoded to retrieve the injected errors.

4.1.3 Verification Environment

To test the functionality of the implemented encoder and decoder, a verification environment is designed as shown in Figure 4.4. The Design Under Test (DUT) is found in the grey box and it consists of the implemented encoder and decoder. The *message* is fed to the Encoder through a transactor1. Transactor1 acts as a controller to the encoder. It will send a message of 239 bytes at a rate of one byte/clock and thereafter, it toggles the *i_enable* signal of the encoder to attach the parity symbols to the message forming a 255 byte-codeword. As the encoded bytes come out of the encoder, they are compared to the *codeword* file generated from MATLAB. Any discrepancy will assert the *tb_enc_error* signal to high.

The encoded codeword, $c(x)$ is corrupted using $e(x)$, the *Error* file from Error Generator designed in MATLAB. The received codeword, $r(x) = c(x) + e(x)$ is then fed into the RS decoder, which knows only $r(x)$ and will calculate $e(x)$. The transmission error,

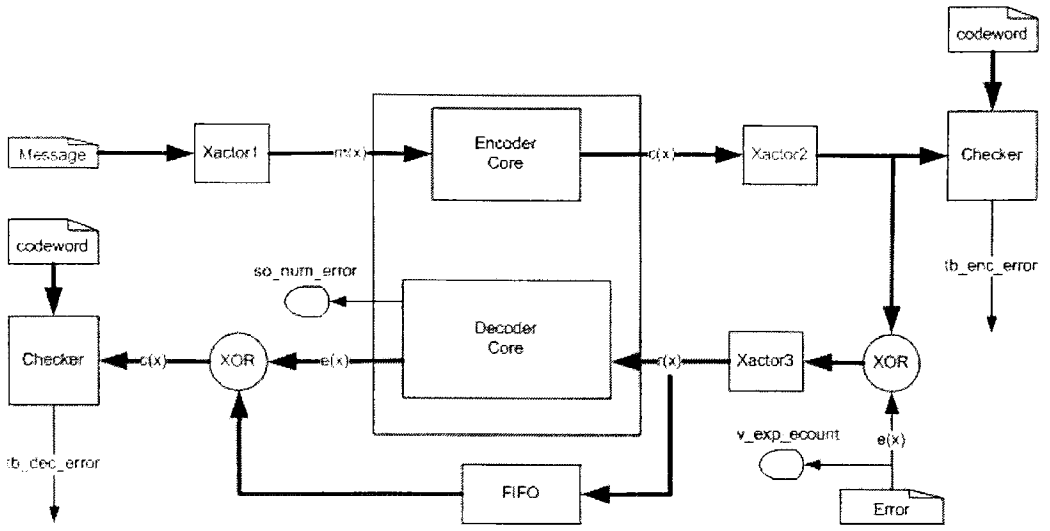

Figure 4.4: System Verification Environment

e(x), which is determined by the decoder, is then added to the delayed received codeword, $r(x)$, to reproduce the transmitted codeword, $c(x)$. The decoded codeword is then verified against the *codeword* file generated from MATLAB. If the decoded codeword does not match the MATLAB codeword, the *tb_dec_error* is pulled high.

4.1.4 Simulation Results

Sixteen different random seeds, ranging from 1 to 255, are used to generate test files with different sets of codewords and errors. Figure 4.5 shows the first 400 us of the simulation. It can be observed that the *tb_enc_error* signal remains low all of the time and this demonstrates that the symbols coming out of the encoder matches the expected symbols from the MATLAB *codeword* file. The encoded symbols, *so_edata*, are then corrupted with *s_error* from the *error* file. For each codeword, the number of errors introduced is recorded. The *v_exp_ecount* displays the number of errors injected from the error file and *so_num_error* signals displays the number of errors corrected by the decoder.

From Figure 4.5, it is observed that the first codeword contains eight errors and is followed by a five-error codeword, and so on. After a latency of 488 clock cycles, the first

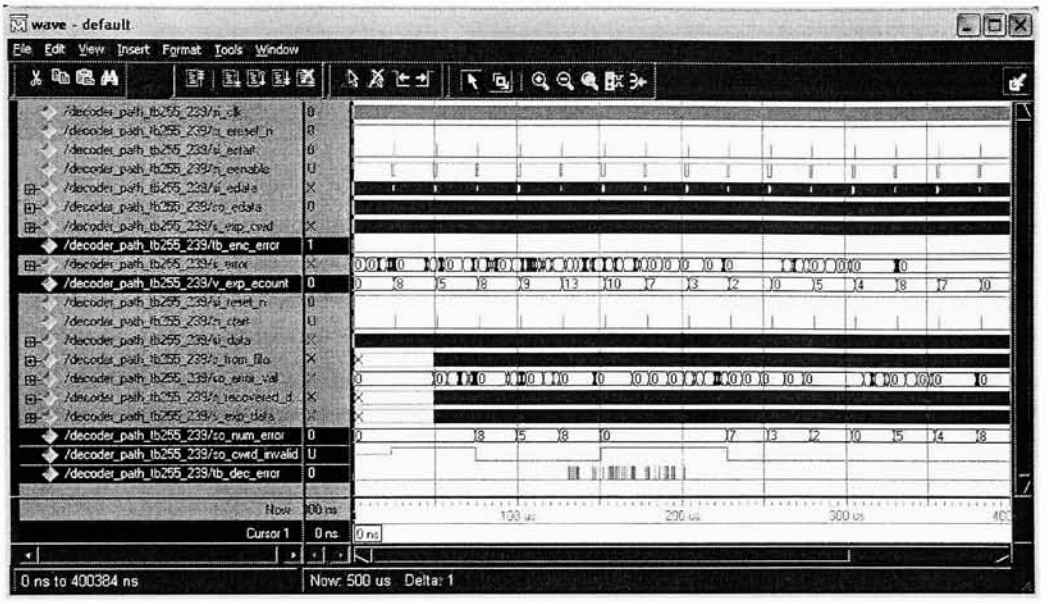

Figure 4.5: Full Range Simulation

decoded symbol comes out of the decoder. The *so_num_error* signal from the decoder indicates that the first decoded codeword contains eight errors, the second contains five errors. However, this signal displays zero for the fourth, the fifth and the sixth codeword, which have nine, thirteen and ten errors, respectively. The decoder can only correct up to eight errors and since these codewords have more than eight errors, the decoder marks the codeword as invalid through the *so_cwrd_invalid* signal. The *tb_dec_error* signal remains low most of the time except for the codeword with more than eight errors. This outcomes of the signals are expected because the received codeword must be discarded. However, regardless the codeword can be corrected or not, the testbench will compare the decoded codeword to the expected codeword from MATLAB. But in this case, the testbench does not know what the expected codeword is. It is comparing the decoded codeword with the expected codeword, had the codeword been correctable. This explains why the *so_cwrd_invalid* signals have been asserted for the uncorrectable codewords.

Figure 4.6 shows the beginning of the encoding process. The first five bytes entering and leaving the encoder are 3, 5, 0, 7, and 4. The *s_error* signal is the error to be introduced

into the codeword and in this example, the first byte contains an error, which is 12. The *si_data* signal represents the received codeword that feeds into the decoder. It is observed that the first byte of the codeword has been corrupted and the first bytes entering the decoder are now 15, 5, 0, 7, and 4.

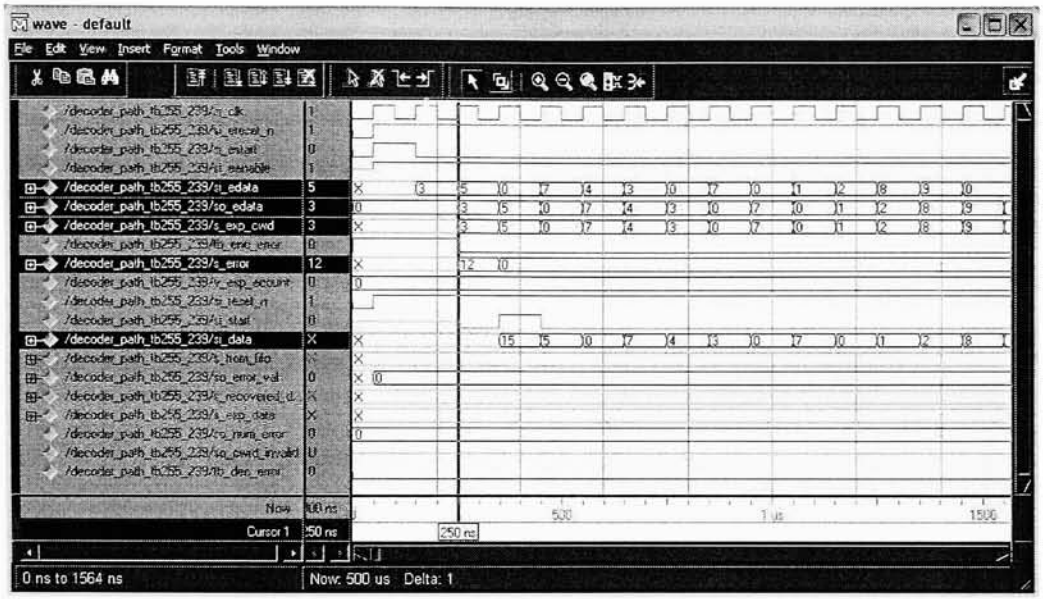

Figure 4.6: First Input Byte

The first byte of the corrected codeword is obtained after 488 cycles, as shown in Figure 4.7. The *s_from_fifo* signal represents the delayed codeword coming out of the FIFO. This signal is added with the decoder output signal, *so_error_val*. The first five bytes of the recovered signal are 3, 5, 0, 7 and 4, which is equivalent to the transmitted codeword.

Another important function that can be observed in Figure 4.7 is the parity attachment. Once the last byte of a message has been sent to the encoder, the *si_eenable* signal is pulled low and the encoder stops reading data from *si_edata* and shifts out the sixteen parity bytes. When generating the 254th symbol of the current codeword, the *si_estart* signal is pulsed to start encoding the next codeword and to append the first byte of the next codeword to the last byte of the current codeword.

One test is arbitrarily designed to test the limit of the decoder. A 16-byte wide burst

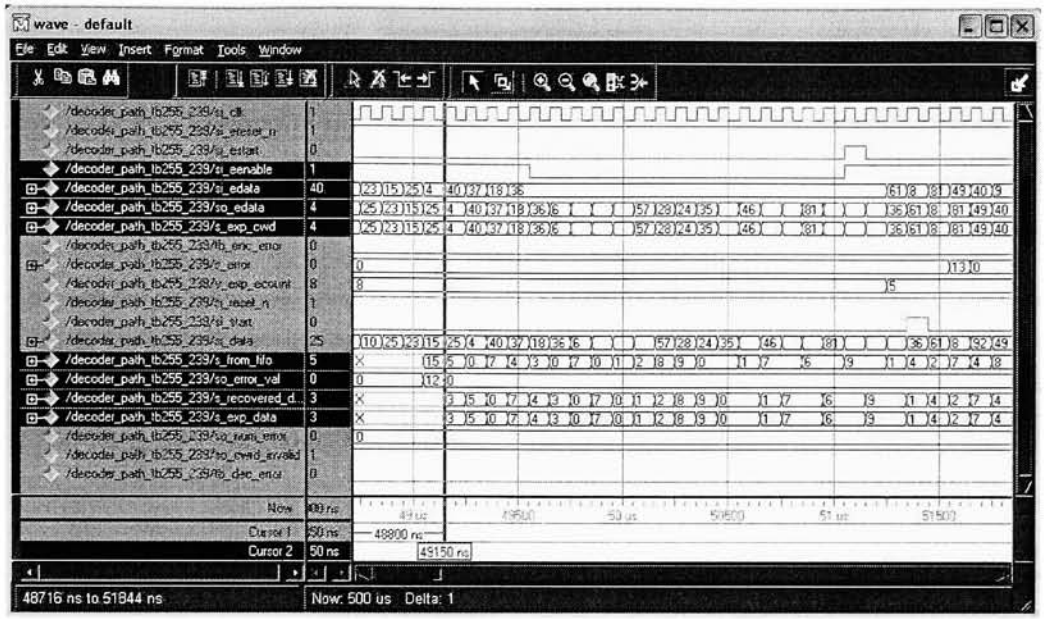

Figure 4.7: First Decoded Byte

error is injected in the stream of data. The burst error is spread over two codewords causing the last consecutive eight bytes of one codeword and the first consecutive eight bytes of the following codeword to be erroneous. The burst errors model the worst case scenario of the error pattern, which could potentially occur in the Optical Networks. After 488 clocks, the correct symbols are generated and compared to the MATLAB generated *codeword* files. The *tb_dec_error* remains low throughout the decoding of these two codewords. The simulation results show that all the sixteen bytes of the two codewords have been recovered by the decoder.

4.1.5 Post-Synthesis Simulation

The synthesized models are generated for Altera's QuartusII and Xilinx's ISE for simulation. In the QuartusII tool, under Settings → EDA Tool Settings → Simulation, the *ModelSim-Altera (VHDL)* option is selected as the Tool name. The two boxes *Maintain hierarchy* and *Generate netlist for functional simulation only* are checked and the design

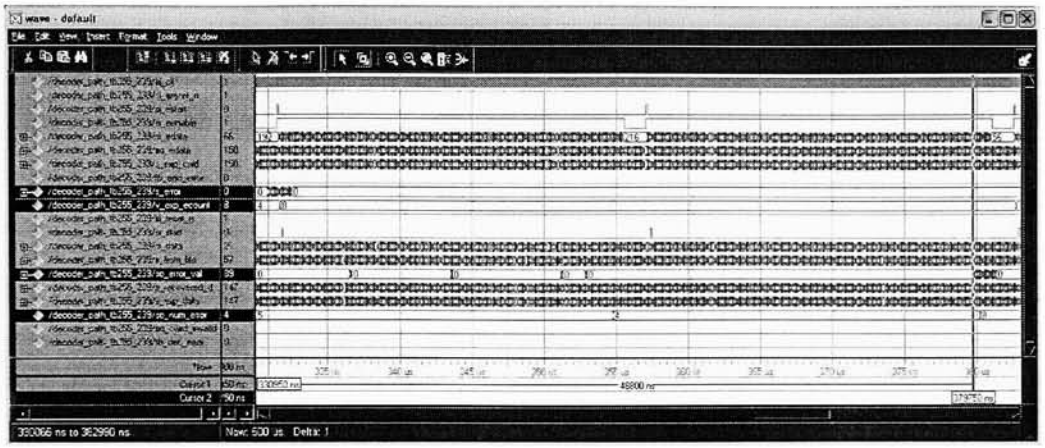

Figure 4.8: Burst Error

is synthesized. The netlist file (RS255_239.vho) is generated in the *../simulation/modelsim*

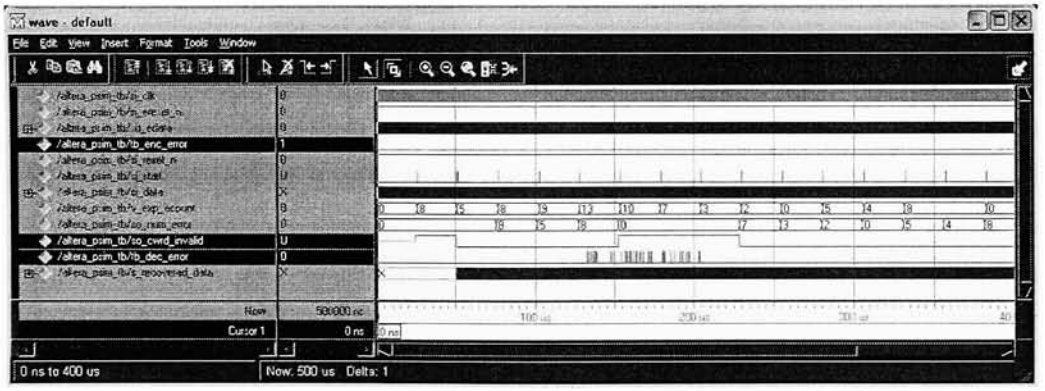

Figure 4.9: Altera Post-Synthesis Simulation

folder, which is created by the EDA Netlist Writer. The generated entity is instantiated in the testbench and simulated in *ModelSim_altera 6.0c*. The simulated results are shown in Figure 4.9.

In the ISE tool, the *Post-Synthesis Simulation Model* is generated with the default option to target ModelSimXE (VHDL). The netlist file (decoder_path_synthesis.vhd) is generated in the local folder. The generated entity is instantiated in the testbench and simulated in *ModelSim XE II*.

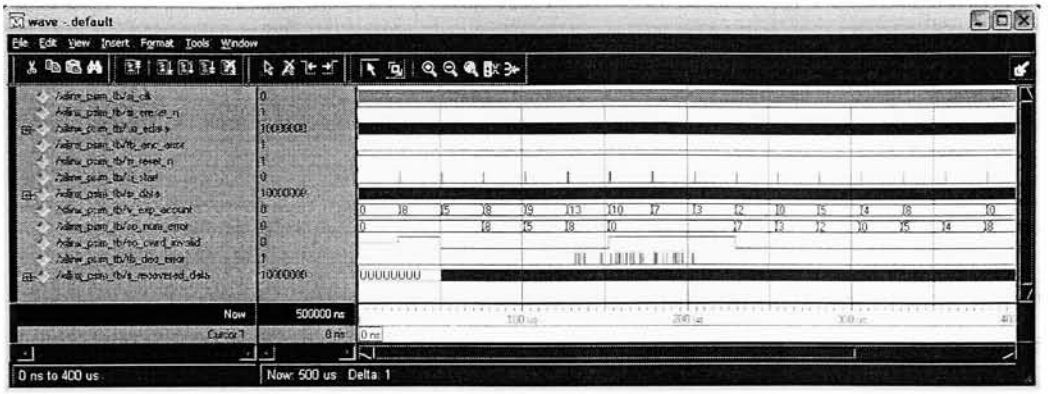

Figure 4.10: Xilinx Post-Synthesis Simulation

From the simulation windows, shown in Figure 4.9 and Figure 4.10, it is observed that the testbench test signals behave similarly to the signals in Figure 4.5. The simulation results demonstrate that the designs are functionally equivalent after being being synthesized using different synthesis tools.

4.2 Synthesis Results

The RS(255,239) codec is instantiated in a top level entity and is wrapped with registers as shown in Figure 4.11. The function of the registers is to prevent the core's primary registers to be potentially mapped into IOBs (I/O Block registers) which would result in a lower logic count. The design is synthesized using Altera's QuartusII v5.0 and Xilinx's ISE v7.1. The QuartusII tool offers three different optimization techniques: Speed, Balance or Area. The ISE tool, on the other hand, does optimization in two steps. The design is synthesized for either area or speed. The synthesized design is then mapped to achieve either the least amount of resource utilization or the highest speed. The trade-off between resource utilization and area is not obvious when implementing OTU-3 FEC [20] in FPGAs. Since the OTU-3 has a specified serial data rate of 40Gbps, a 32 byte data bus is used at a speed of 169MHz. To satisfy this requirement, not only the RS decoder needs to run at a speed of 169 MHz, but it needs to be small enough to fit a total of 32 decoders in one FPGA. The

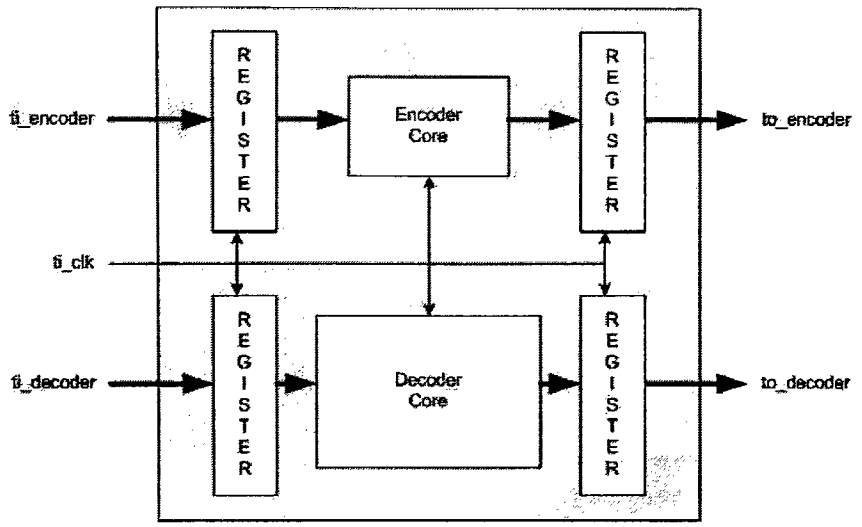

Figure 4.11: Core Wrapper

following section goes over how to choose a suitable device to implement FEC for OTU-3.

4.2.1 Device Selection

Altera provides a wide device selection from the StratixII family. Eight devices are selected as the targets for synthesis. For each device, the RS decoder is synthesized with a timing constraint of 5.9 ns for all three optimization techniques; speed, balance and area. The unit of measure for resource utilization in StratixII is the *ALUT* [23]. There are two ALUTs in one Adaptive Logic Module (ALM). The ALM is the basic building block of logic in the StratixII architecture. One ALM is made up of two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, and a register chain, which allows the ALM to efficiently implement various arithmetic functions and shift registers.

The logic resource utilization of one decoder is represented as a percentage of the total logic resources available in a specific FPGA. To approximate the FPGA logic resources needed to implement FEC in OTU-3, the percentage of resource utilization of one decoder is scaled up by 32 times. The synthesis results are shown in Figure 4.12. Each device has a set of three data points, which confirms that high speed performance is achieved at

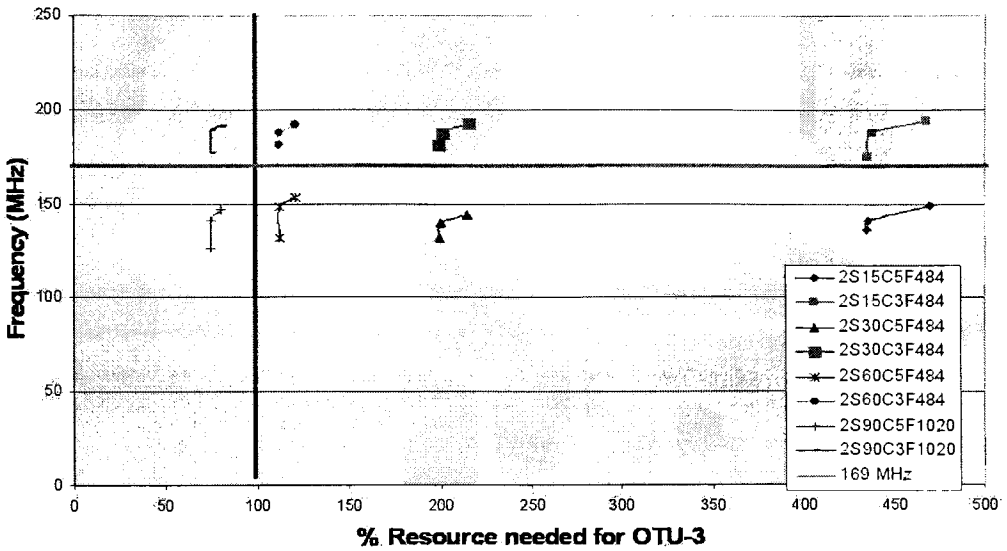

Figure 4.12: Synthesis of RS decoder on StratixII

the expense of hardware resources. The plot is divided into four regions; a horizontal line to represent a threshold frequency of 169 MHz and a vertical line at 100% representing that the entire decoder will use up the total available logic resources in one FPGA. Among the selected StratixII devices, the graph shows that only the $2S90C3F1020$ device can be used to fit 32 RS decoders running at a frequency of 169MHz. The optimization technique "optimizing for area" is opted because not only the synthesized design utilizes the least amount of resources, but also meets the required frequency.

Similar methodologies are adopted to select a Virtex4 device from Xilinx. Among the three available platforms, six devices are selected from the Virtex4-FX platform due to its Rocket I/O [24] feature. The synthesis flow in ISE is different from QuartusII's. On each of selected Virtex4 devices, the design is synthesized using four optimization techniques, which are "Synthesized for area and mapped for area", "Synthesized for area and mapped for speed", "Synthesized for speed and mapped for speed," and "Synthesized for speed and mapped for area." The Virtex4's basic building block is called the Configurable Logic Block (CLB) [25]. One CLB consists of two pair of *Slices* (SliceL and SliceM), and one *Slice*

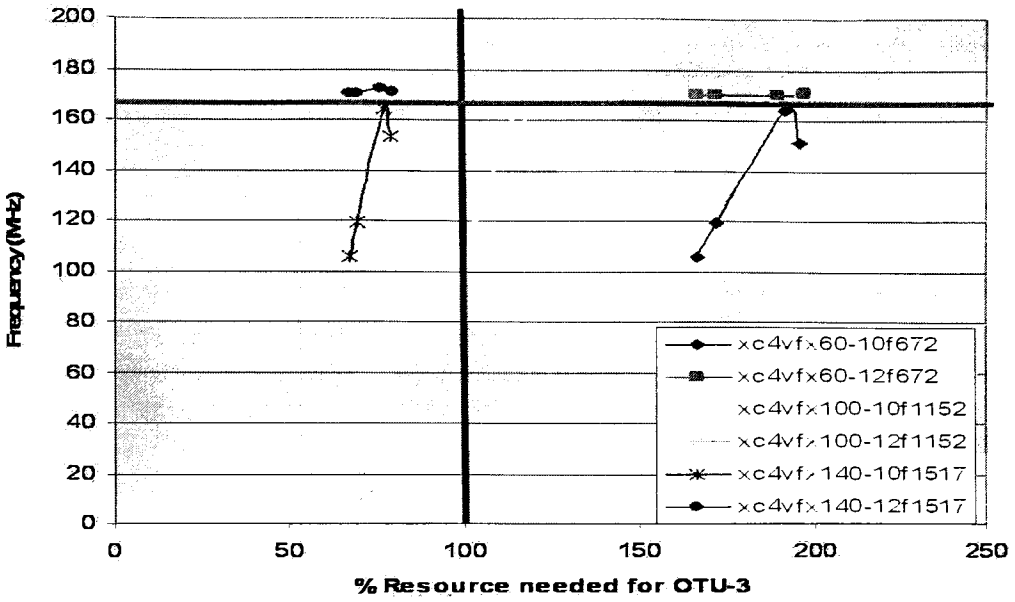

Figure 4.13: Synthesis of RS decoder on Virtex4

is the unit of measure for the logic resource utilization. Each slice contains two function generators, two storage elements, some arithmetic logic gates and fast carry look-ahead chain. SliceL is used as logics only, and SliceM can be configured as logic, distributed RAM or shift register.

The synthesis results are displayed in Figure 4.13, and reveal that among the selected Virtex4 devices, only the $xc4vfx140 - 12f1517$ device can potentially be used to implement RS decoder to support FEC in OTU-3. Among the four selected optimization techniques in ISE, "Synthesized for area and mapped for area" optimization technique is chosen to obtain a design with the least amount of slices while respecting the specified timing constraint. When comparing Figure 4.12 to Figure 4.13, it is observed that the StratixII devices exhibits the same behavior irrespective they meet timing or not. However, the Virtex4 devices behave differently. This is due to the difference in the constraint and timing analysis philosophies of QuartusII and ISE. By default, the QuartusII tool analyzes and optimizes all possible paths, disregard of whether they are constrained or not. However, the ISE tool focuses only on constrained paths and does not optimize or report unconstrained paths. To

test for the maximum performance of the proposed decoder on Virtex4, the threshold of 169MHz needs to be raised until the design does not meet the timing requirement.

4.2.2 Resource utilization by individual decoding block

Once the optimization technique for each synthesis tools has been selected, the percentage logic resource utilization of individual functional block relative to one RS decoder is analyzed on both $2S90C3F1020$ and $xc4vfx140 - 12f1517$. The pie charts in Figure 4.14 represent the percentage utilization of each block in one RS decoder in both StratixII and Virtex4 device. It can be observed that the percentage of resources allocated to the *Key Equation* in Virtex4 is more than that in StratixII. The reason behind this difference is the use of memory to implement the look-up table. Xilinx's ISE tool translates the Key Equation's look-up table into Read-Only Memory (ROM), which is implemented using logic blocks. However, the StratixII architecture allows Altera's QuartusII tool to pushes the look-up table into the memory, resulting in a less logic resource utilization. Another obser-

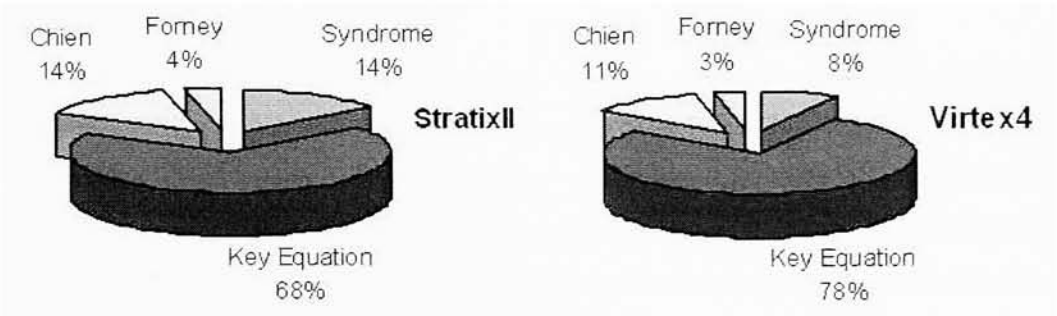

Figure 4.14: Relative Functional Block Utilization

vation is the difference in the percent logic utilization between the *Chien* and the *Syndrome* blocks. The distribution in Virtex4 is expected and reflects the VHDL code in terms of the number of logic blocks. However, in StratixII, both blocks share the same percentage of logic utilization. This is the result of "register packing," which is a feature in StratixII that allows the device to utilize the unused register or combinational logic of one ALM for unrelated functions.

4.2.3 Benchmark

The optimized version of the proposed RS decoder is then compared to Altera's decoder on six StratixII devices. Using the *MegaWizard Plug-In Manager*, Altera's Reed-Solomon decoder v4.0 [21] is generated and instantiated in a top-lever wrapper. The optimization technique is set for *area* with a timing constraint of 5.9ns. Figure 4.15 shows the compari-

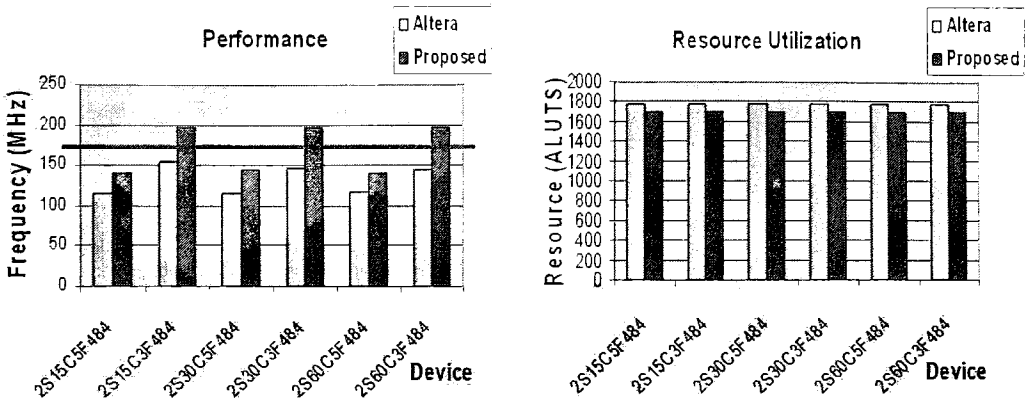

Figure 4.15: Altera's vs Proposed Decoder

son between Altera's decoder and the proposed decoder. The graph on the left presents the high frequency capability of the proposed decoder in contrast with Altera's RS decoder. The targeted frequency was achieved only on the fast StratixII device while the resource utilization was independent on the device selection. The graph on the right represents the logic resource utilization of both decoders. It is observed that the proposed implementation utilizes less logic resources than Altera's core, independent of the speed grade. The same trend follows for memory utilization too. Altera's decoder utilizes 14336 memory bits while the proposed decoder utilizes 4096 memory bits.

The portable RS decoder is then compared to Xilinx's RS decoder. Using the *Core-Gen & Architecture Wizard* tool, the Xilinx's RS(255,239) decoder [22] is generated and instantiated in a top-level wrapper. Before synthesizing the designs, the optimization goal for synthesis and the optimization strategy for mapping were set for area with a timing constraint of 5.9ns. The collected data are presented in Figure 4.16. The required speed

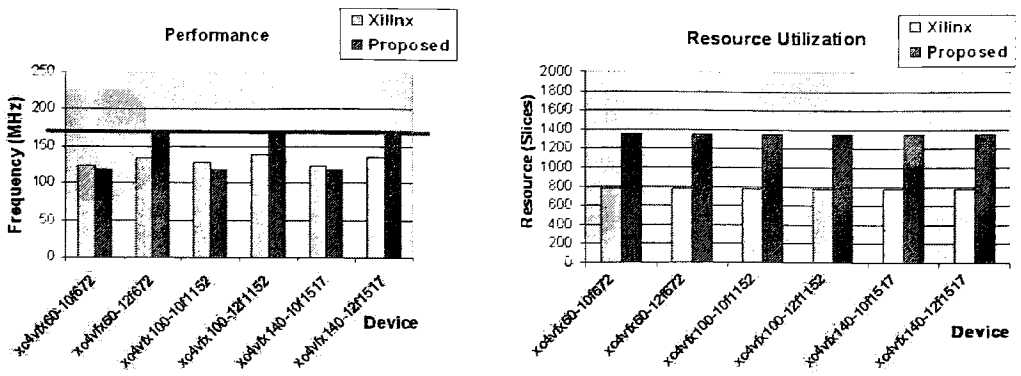

Figure 4.16: Xilinx's vs Proposed Decoder

to support FEC in OTU-3 [20] is met by the proposed decoder targeting the fast device as shown on the left side graph of Figure 4.16. The plot on the right represents the logic resource utilization of both decoders on selected Virtex4 devices. The synthesis results show that the Virtex4 architecture favors Xilinx's RS decoder in terms of logic resource utilization. Xilinx's decoder is utilizing about half the logic resources utilized by the proposed decoder. However, in terms of the memory resources, Xilinx's decoder is using 2 RAM blocks while the proposed implementation is utilizing only 1 RAM block out of 232 blocks in fx60, 376 blocks in fx100 and 552 blocks in fx140. One major difference between Xilinx's decoder and the proposed decoder is the implementation of the *Key Equation* [4]. Xilinx implements the Euclidean algorithm [32] to solve the *Key Equation* while the proposed decoder uses the Berlekamp-Massey algorithm [26]. The results are "unexpected" because the research [31] in ASIC suggested that the BM algorithm consumes less resources than the Euclidean algorithm. It seems that the argument is not valid on FPGA. Depending on the device architecture, the Euclidean algorithm can utilize less logic resources than the Berlekamp algorithm as shown in the Resource Utilization graph of Figure 4.16.

The results show that the proposed decoder can support FEC in OTU-3 if it is implemented in a fast speed grade and high density FPGA. The smallest StratixII device that could be used to implement FEC is the $2S90C3$ device. As for Virtex4, the only device from the FX family that could used is the $xc4vfx140-10$ device. These two FPGA devices

were selected among the others after synthesizing the design on each device with different optimization technique. It was found that these two devices would fit the 32 RS decoders needed to support FEC at a rate of 169 MHz. Moreover, it was observed that on Virtex4, the Euclidean algorithm, which is known to consume more resources than Berlekamp algorithm on ASICs, actually utilizes less logic resources than the Berlekamp algorithm. This "unexpected" results lead to the future work, which involves designing a Euclidean-based RS decoder to confirm the performance of Euclidean's algorithm on FPGA devices.

Chapter 5

Conclusion & Future Work

More than 80% of the world's long-distance and data traffic is carried over fiber-optics, ranging from global networks to desktop computers. To increase the reliability of data, the concept of Forward Error Correction (FEC) is adapted. FEC is a technique used to correct transmission errors at the receiving side. FEC is highly used in OTN G.709 protocol [20]. The protocol defines three standard interfaces, OTU-1, OTU-2 and OTU-3, providing performance and facilitating evolution to higher backbone bandwidths. The OTU-3 standard interface, which has the highest data transfer rate of 43 Gbps, is still under development. To achieve a rate of 43 Gbps, a databus of 32 bytes clocking at 169MHz is used.

Since FPGA has become very popular in industrial product development, we proposed an FPGA implementation of the RS(255,239) codec running at a minimum frequency of 169MHz. As expected, the Key Equation Solver block was the bottle-neck of the RS decoder in terms of both speed and logic resource utilization. Pipelining the critical path and serializing logics through resource sharing helped in improving the timing and resource utilization. The optimization did result in an average speed up of 100% and an average reduction of 13% in logic resource utilization. The synthesis results showed that the proposed decoder could operate at a minimum frequency of 169MHZ when targeting the high-speed grade devices. Moreover, an FPGA with a high logic count shall be selected to implement FEC in OTU-3.

Research work in the literature has suggested that the Berlekamp-Massey algorithm consumes less resources than the Euclidean algorithm on ASICs. However, this argument

may not be valid on FPGAs. When developing in FPGAs, the performance does not only depend on the design of the system, but also on the architecture of the targeted device. Xilinx's RS decoder, which implements the Euclidean algorithm to solve the *Key Equation*, utilizes almost half the logic resources utilized by the proposed decoder, which implements the Berlekamp-Massey's algorithm. The Virtex4 architecture seems to favor the Euclidean algorithm. However, it is not clear if the low logic count and block memory utilization of the Euclidean algorithm is consistent on all FPGAs or valid only on Virtex4. The future work will involve designing a RS(255,239) decoder using the Euclidean algorithm to solve the *Key Equation* to generate the error-locator and error-evaluator polynomial. The research must be geared toward analyzing the algorithm to see which part could be implemented using memory blocks instead of logic blocks. The Euclidean-based RS decoder must be able to run at a minimum frequency of 169MHz to support FEC in OTU-3. The performance and resource utilization of the Euclidean-based RS decoder will then be measured on both StratixII and Virtex4 devices. The data will confirm how the algorithm used to implement the *Key Equation Solver* depends on the FPGA architecture.

Bibliography

[1] E.R Berlekamp. *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-correcting Coding and decoding: Turbo-codes. *IEEE International Conference on Communications*, 2:1064–1070, May 1993.

[3] S.R. Blackburn and W.G. Chambers. Some remarks on an algorithm of fitzpatrick. *IEEE Transactions on Information Theory*, 42(4):1269–1271, July 1996.

[4] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge Publishers, 2003.

[5] M.H. Cheng. Generalised berlekamp-massey algorithm. *IEE Proceedings - Communications*, 149(4):207–210, August 2002.

[6] R. T. Chien. Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. *IEEE Transactions on Information Theory*, IT:357–363, October 1964.

[7] Anh Dinh and Daniel Teng. Design of a high-speed rs(255,239) decoder using 0.18m cmos. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, May 2004.

[8] A. Drukarev and Jr D. Costello. A Comparison of Block and Convolutional Codes in ARQ Error Controle Schemes. *IEEE Transactions on Communications*, 30(11):2449–2455, November 1988.

[9] P. Fitzpatrick. On the key equation. *IEEE Transactions on Information Theory*, 41(5):1290–1302, Septempter 1995.

[10] A. Flocke, H. Blume, and T. G. Noll. Implementation and modeling of parametrizable high-speed reed solomon decoders on FPGAs. *Advances in Radio Science*, 3:271–276, March 2005.

[11] G. D. Forney. On decoding bch codes. *IEEE Transactions on Information Theory*, IT:549–557, October 1965.

[12] Charles Constantine Gumas. Turbo codes rev up error-correcting performance. *Personal Engineering and Instrumentation News*, pages 61–66, January 1998.

[13] A. Haase, C. Kretzschmar, R. Siegmund, D. Muller, J. Schneider, and M. Langer. Design of reed solomon decoder using partial dynamic reconfiguration of xilinx virtex fpgas - a case study. In *Proceedings of the IEEE International Conference on Design, Automation and Test*, March 2002.

[14] Eui-Seok Kim and Yong-Jin Jeong. A New Finite Field Division (FFD) Algorithm and its Hardware Architecture. *accessible via http://rta.gwu.ac.kr/publications/euiseok/2004_10_Finite%20Field%20Division.pdf*, October 2004.

[15] Hanho Lee. Modified euclidean algorithm block for high-speed reed-solomon decoder. *Electronics Letters*, 37(14):903–904, May 2001.

[16] Hanho Lee. A vlsi design of a high-speed reed-solomon decoder. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, 2001.

[17] Hanho Lee. An area-efficient euclidean algorithm block for reed-solomon decoder. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, May 2003.

[18] Hanho Lee. A high-speed low-complexity reed-solomon decoder for optical communications. *IEEE Transactions on Circuits and Systems II*, 52:461–465, August 2005.

[19] Hanho Lee and A. Azam. Pipelined recursive modified euclidean algorithm block for low-complexity, high-speed reed-solomon decoder. *IEE Electronics Letters*, 39:1371–1372, September 2003.

[20] Manual. Interfaces for the Optical Transport Network (OTN), March 2003.

[21] Manual. Reed-Solomon Compiler User Guide, January 2003.

[22] Manual. Reed-Solomon Decoder v5.1, April 2005.

[23] Manual. Stratix Device Handbook, May 2005.

[24] Manual. Virtex-4 User Guide, September 2005.

[25] Manual. Virtex-4 Family Overview, February 2006.

[26] J.L Massey. Shift register synthesis and bch decoding. *IEEE Transactions on Information Theory*, Vol. IT-18(1):196–198, January 1969.

[27] White Paper. A G.709 Optical Transport Network Tutorial, October 2003.

[28] Arun Raghupathy and K. J. R. Liu. Algorithm-based low-power/high-speed reed-solomon decoder design. *IEEE Transactions on Circuits and SystemsII: Analog and Digital Signal Processing*, 47(11):1254–1269, November 2000.

[29] Irving S. Reed and Gustave Solomon. Polynomials codes over certain finite fields. *SIAM*, Vol 8(2):300–304, 1960.

[30] Frederic Rivoallon. Achieving Breakthrough Performance in Virtex-4 FPGAs, May 2005.

[31] Dilip V. Sarwate and Naresh R. Shanbhag. High-speed architectures for reed-solomon decoders. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, 9(5):641–655, October 2001.

[32] Y. Sugiyama, Y. Kasahara, S. Hirasawa, and T. Namekawa. A method for solving key equation for goppa codes. *Information and Control*, Vol 27:87–99, 1975.

[33] International Telecommunication Union. *ITU-T G.709*, February 2001.

[34] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimim decoding algorithm. *IEEE Transactions on Information Theory*, 13(11):260–269, April 1967.

[35] Stephen B. Wicker. *The Decoding of BCH and Reed-Solomon Codes*. Prentice Hall, 1995.

[36] Stephen B. Wicker and Vijay K. Bhargava. Reed-solomon codes and their applications. In *Proceedings of IEEE 18th Annual Workshop on Computer Communications*, 1994.